IEEE MICRO

APRIL 1989

Chips, Systems, Software, and Applications

Designing for upward compatibility

AT&T's WE32200

Motorola's 88000

IEEE COMPUTER SOCIETY

THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC

Cover: Design & Direction

IEEE MICRO

Volume 9 Number 2 (ISSN 0272-1732) April 1989

Published by the IEEE Computer Society

Departments

Feature Articles

MULTIPLY THE BENEFITS OF CASE

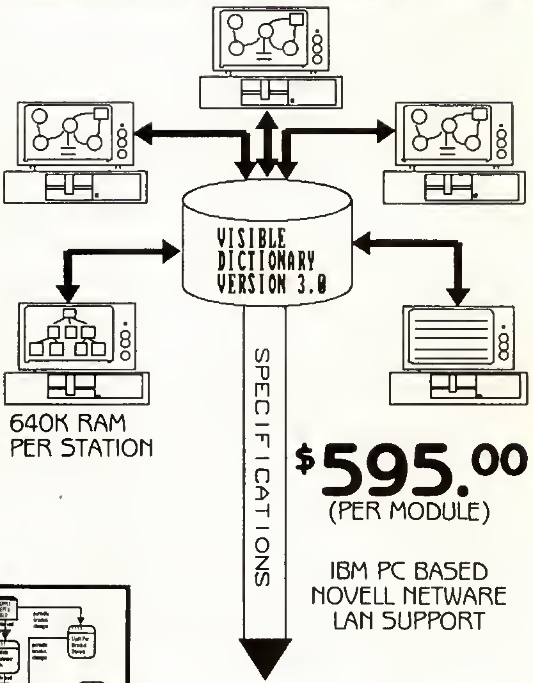
with the LAN BASED VISIBLE SOLUTION®

Coherent project development builds high quality systems faster!!!

All project team members use the same data dictionary!!!

Methodology based expert rules checks all work for consistency across the entire project!!!

Flexible, friendly, easy-to-use graphical interface brings project staff up-to-speed quickly!!!



640K RAM PER STATION

$595.00
(PER MODULE)

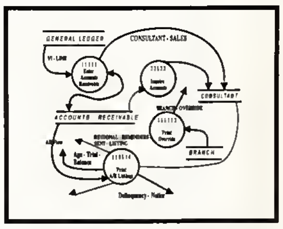
IBM PC BASED NOVELL NETWARE LAN SUPPORT

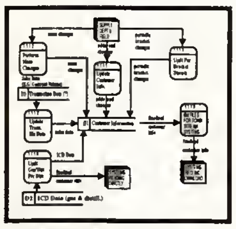
VISIBLE ANALYST® WORKBENCH

THE NETWORKABLE VERSION 3.0--NEW FEATURES

identical look & feel as single user version • Configurable executable modules • Unlimited number of potential users on the LAN • Little or no system degradation resulting from Workbench use • Project wide rules validation • Multiple concurrent users • Common data dictionary • Full file and record locking • Multiple security levels •

Ever since computer aided software engineering products came on the market, interest has been high. And so, not suprisingly, has been the cost of owning them. But now Visible Systems is offering state of the art CASE technology at an incredibly affordable price. $595.00 per module. These modular tools give your company the ability to grow in CASE power. The modules of the Workbench build upon each other to provide more and more productivity.


YOURDON DIAGRAM


GANE & SARSON

VISIBLE ANALYST - MODULE A

The Visible Analyst is the foundation of the Workbench. The user can create any analysis or design diagram. Multiple line styles, text fonts, and symbol types are standard features. Other features include a custom symbol generator and a reuseable constructs library. Build and store entire diagrams for reuse in other projects. Boilerplates allow diagrams to be customized. Diagrams can be output in presentation quality using HP laser printers. Entire subsystems or single diagrams can be moved from one project to another while maintaining nested relationships. The LAN version allows multiple analysts to build or edit diagrams, within the same project, concurrently!!

VISIBLE RULES - MODULE B

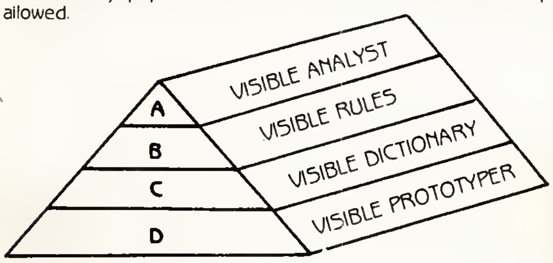
The Visible Rules is the on-line expert system that makes the use of structured techniques possible. The rules module implements either of two powerful, methodology based, expert rules systems: Yourdon/DeMarco or Gane & Sarson. Once enabled, the rules system monitors the diagramming process for naming conventions, data balancing, proper symbol usage etc. After diagrams have been built, the rules module analyzes them for errors. Error reports are output to screen or hardcopy. And the analysis takes place on-line. One never leaves the diagram. Problem resolution and identification is faster and easier. Logical data flow splitting is fully supported. Dictionary is automatically populated. Unlimited levels of nested decompositions are allowed.

VISIBLE DICTIONARY - MODULE C

Visible Dictionary is the central data repository for the system being modeled. It is multi-user and interactive. Many users may access it simultaneously. Full file and record locking capabilities keep users from colliding and corrupting data. Visible Dictionary has many powerful features including: "where used" listings, wild card searches and finds, full import/export, automatic entry generation and entry updating, on-line accessability from diagrams, many different entry types, customized report formatting, and much more. The Dictionary organizes, reports, and categorizes all project data. File format information is provided for data transfers in ASCII format. The specification developed using the Visible Workbench becomes a company asset that can be used to build upon in later stages of the systems development lifecycle.

***NEW!!!* VISIBLE PROTOTYPER - MODULE D**

The Visible Prototyper is our new screen design and full system simulation tool. Four levels of prototyping are supported: Screen design and panel linking; Panel branching, Data selection; and System simulation. User interface is "pre-approved". Integrates with all design methodologies, use for education and training -reduces costs. The Prototyper is available as a standalone or integrated fourth module with links to the Visible Dictionary for consistency in data names.



Visible Systems CORPORATION
49 Lexington Street • Newton, MA 02165 • (617) 969-4100
FAX (617) 969-9505 Telex 261102 VSCUR
LEADING EDGE CASE TECHNOLOGY SAME DAY SHIPMENTS WORLDWIDE

Circulation: *IEEE Micro* (ISSN 0272-1732) is published bimonthly by the IEEE Computer Society, 10662 Los Vaqueros Circle, Los Alamitos, CA 90720-2578; IEEE Computer Society Headquarters, 1730 Massachusetts Ave., NW, Washington, DC 20036-1903; IEEE Headquarters, 345 East 47th St., New York, NY 10017. Annual subscription: $18 in addition to IEEE Computer Society or any other IEEE society member dues; $33 for members of other technical organizations. Back issues: $10 for members; $20 for nonmembers. This journal is also available in microfiche form.

Postmaster: Send address changes and undelivered copies to *IEEE Micro*, 10662 Los Vaqueros Circle, Los Alamitos, CA 90720-2578. Second class postage is paid at New York, NY, and at additional mailing offices.

From the Editor-in-Chief

This issue

Those of you who follow the *IEEE Micro* editorial schedule will be expecting this issue to feature the latest in research and development from the Far East. Due to the switchover in the Editor-in-Chief position and my failure to communicate, we did not receive the articles from guest editor Ken Sakamura in time to do a proper job of editing for this issue. We will take steps to ensure that this will not happen again. I can assure you that it will be worth the wait until June for these articles.

This issue contains five very interesting and diverse papers. An article by editorial board member Victor Huang and associates from AT&T discusses the procedure and the thought process that takes place when redesigning and upgrading new hardware. Charles Melear discusses the 88000 series of processors and the various aspects of the computing capabilities of these powerful RISCs.

An article by Ioan Dancea presents a procedure and the necessary way to implement a dynamically changeable microcomputer interface using either software or hardware. The concepts presented by Dancea can be extended to the development of a Boolean processor. In the area of software tools Mike Papazoglou discusses the implementation of a new database management system for small- to medium-size computer systems.

Bernd Ingenbleek, Klaus Woelcken, and Claudia Matthaus present a different way of looking at the simulation problem for very complex circuits. They use graph theory to decouple the network equations for fast, efficient calculation. Finally, Richard Mateosian returns to *Micro* as a contributing editor of the Micro Review department, and it is a pleasure to have him back on board.

This past January I participated in a meeting to discuss the future of the IEEE Computer Society publications. We made plans to address the ways Computer Society publications could best support the present and future needs of its members. One of the topics under discussion was *IEEE Micro*: how it is viewed and where it fits into the Computer Society publications. While very little question existed about *Micro*'s status in the Computer Society, we felt it would benefit *Micro* to have a little better definition in its title. To this end, on the front cover you will find a subtitle added to the current title. This subtitle (Chips, Systems, Software, and Applications) reflects the nature, the philosophy, and the character of *IEEE Micro* in its present form and will serve our magazine well in the future.

J. Hootman

In the mailbag

December 1988 issue

"I liked new products and letters to the editor. I would like to see more tutorials." J.J.V., Miami Beach, FL (Marlin Mickle has done an excellent job with new products; we always welcome letters to the editor. We are in the process of getting some tutorial material for *Micro*, and you should be seeing it in the near future.—J.H.)

"I like the honesty of J. Farrell and R. Mateosian. . . . I disagree with the statement that the decline in the subscriptions to optional publications or the Computer Society is due to the decline in the economy. I would like to see reviews of database v. spreadsheets. . . ." P.C., Culver City, CA (Jim Farrell and Richard Mateosian are two of the most honest men that I have had the privilege to work with. The decline in subscriptions can be traced to many things: competition, the cut in tax write-offs for professional publications, and the decreased support of companies that subscribe to publications for their employees. And, last but not least, there is just a lot of material and information out there to read and to learn. We would be happy to publish a quality article on the evaluation of popular, new commercial spreadsheet and database management software.—J.H.)

"I liked the DSP issue. I would like to see a DSP article related to a fast control system." D.L., Milwaukee, WI (Guest editors Stephen Dyer and Robert Morris and the authors are to be commended on the excellent DSP issue.—J.H.)

"I would like to see more reports about graphics processors." S.K., Graz, Austria (We have an article in the queue on the use of a signal processor as a graphics processor, and it will be published soon.—J.H.)

"This was a particularly excellent issue; all the articles were timely and well written. I would like to see more articles on numerical methods and general-purpose how-to information." P.N., Arlington, MA

"I liked the entire issue. Would like to see literature on Motorola's DSP and the 96000 series processors." W.J.F., Cambridge, MA (I have forwarded your request for literature to Motorola.—J.H.)

"I liked the in-depth coverage of the currently popular DSPs." C.D., Whittier, CA

"I liked specific applications sample programs for the TMS320C30. I would like to see other applications using the TMS320C30." R.C., Bristol, RI

"Your magazine is the best of IEEE." A.M., Tehran, Iran

"I liked comparative articles on DSP chips, which are excellent. New products are very interesting, especially thin-card memory." J.M., Westwood, MA

"To Jim Farrell: Thanks for your excellent job. You will be sorely missed by a lot of readers. I hope your successors learn your way of getting to the readers." A.R.B., Mexico City

October

"I liked the piece on neural nets, more please." E.R., New York

"I liked all the departments and the special-feature article on benchmarking. I would like to see more on neural networks." J.O.S., Makati, Metro Manila (We are willing to publish quality articles, either tutorial or research in nature, on neural nets. A paper on the hardware implementation of neural nets would make a significant contribution to the general area of neural net research and development.—J.H.)

"I would like to see the application of microprocessor systems in medical instrumentation." S.S., Bandung, Indonesia

"I liked the Unix benchmarking article." B.G., Walpole, West Australia

"I liked the accurate Unix benchmarking, Mutabor, ASP, and advanced database accelerator (articles). I would like to see more about C++; generally, object-oriented programming, expert systems, concurrent programming, and algorithms." D.T., Bucharest

August

"I liked New Products and Product Summary. I disliked Micro Law and the cover. I would like to see more articles on system design (operating systems and telecommunications systems), software algorithms, and advertisements." G.L., Hong Kong

"Good magazine, broad appeal." T.B., Vancouver, British Columbia

"I liked the cache memory article." A.V., Santa Cruz(e), Bombay

---

HOT Chips Symposium

Stanford University, Palo Alto, California
Monday & Tuesday, June 26-27, 1989

The first HOT Chips Symposium will bring together researchers and developers of chips used to construct high performance workstations.

Featuring:

Informal format
Presentations, not papers
Interaction with speakers
Copy of presentations
Lodging on or near the Stanford Campus

Sun SPARC
MIPS R3000
Intel CPUs 486, i860, 80960CA
TI Graphics 34020/82
Many others...

Registration:

	Pre-Reg.	On-Site
IEEE or Computer Society Member	$145	$180
Non-Member	$180	$225
Full-Time Student	$65	$100

Information:

Call, write or FAX (415/941-6699)
Dr. Robert Stewart, General Chair
Stewart Research Enterprises
1658 Belvoir Drive
Los Altos, CA 94022

Sponsored by the Technical Committee on Microprocessors and Microcomputers of the IEEE Computer Society

IEEE COMPUTER SOCIETY

IEEE THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC

Reader Service Number 2

Micro World

Hubert Kirrmann
Asea Brown Boveri Research Center
CRBC.1
CH-5405 Baden, Switzerland

Fault-tolerant computing in Europe

Fault-tolerant computing always fascinates computer engineers and users. This fascination probably results from traumatic experiences with personal computers. Replacing time-proven analog and electromechanical devices with unpredictable microprocessors leaves the engineer with a great responsibility and some doubts. However, the simple fact that fault-tolerant computers exist calms anxious users. One entertains the illusion that computers could never fail if one is just willing to spend more.

Contrary to popular belief, fault tolerance is not a substitute for quality. You can build a bridge through massive use of rotten beams, but you may find that one with a few sound beams is more reliable. Top quality is the condition for replication. Therefore, the first step to fault tolerance is fault intolerance, that is, fault avoidance. We achieve fault avoidance by careful design, production quality, thorough debugging, and extensive testing. Still, residual errors always remain.

When in spite of quality efforts the reliability goals still cannot be achieved, redundancy may be introduced. This drastic measure boosts the price by a factor of three to six times with respect to the simplex system—not only in the duplication of hardware costs but also in all auxiliary functions for communication, spares updating, state saving, and so on. Duplication does not exempt us from checking, either. Lurking faults can defeat any redundancy—it makes no sense to carry a spare tire if you do not know it is good. Since redundancy is costly, we introduce it only where economically interesting or where the regulation authorities impose it.

In high-availability applications, like telecommunication or energy distribution, a fault-tolerant computer must pay for itself through the downtime it spares. A duplicated computer may provide from 10 to 100 times less downtime than the simplex one. But, duplication only becomes interesting if computer downtime is an important fraction of total plant downtime. In a newspaper press, for instance, computer downtime can become negligible in the face of factors like plate reloading, paper tears, and union strikes.

Contrary to popular belief, fault tolerance is not a substitute for quality.

In safety applications, the question is not any more *if* you should use redundant computers, *you must*. In railway, air traffic, or boiler control, the regulation authorities have been reluctant to accept computers. They understand fairly well how a relief valve can fail, but failure of a computer appears unpredictable to them—it often is also for its designer. Further, most safety devices, being purely passive, require no auxiliary energy to operate. The introduction of microprocessors is therefore a risk, and the regulating authorities often impose redundancy.

It makes no sense to replicate the hardware if the software is not rock-stable. In fact, this point has turned out to be one of the major problems in certification. Some advocate the use of diverse software, that is, letting two or more independent teams write N versions of the same software and compare the results. Others argue that it is better to write the software just once and spend the same amount of money to test it. And still others argue that well-known software in massive use (such as the IBM PC's MS-DOS) could be a better base for fault tolerance than operating systems written for some obscure redundant computer. Software fault tolerance is far from being a science; it is an act of faith. This point may explain the bitter discussions between sceptics and believers of different credos, N-version programmers, and other block recoverers. It is truly a research topic.

Fault tolerance is a psychological business. One does not sell a computer, but confidence. Many manufacturers of fault-tolerant computers offer the same system as both a simplex and a redundant machine. The users are attracted by the option that if reliability is insufficient, they can upgrade their system to

The European flag

fault tolerance. After looking at the price tag of the duplicated machine, they usually find out that their application is not as critical as they thought, and they settle for the simplex machine. So, fault tolerance is also a good sales argument. In one particular case, for 100 installed machines, 98 fall into the simplex type.

The disciplines related to fault-tolerant computing all involve some kind of redundancy in hardware or software: online testing, error detection-correction, redundant hardware, and diverse programming. We don't usually consider software quality assurance, development tools, and off-line testing or other methods for fault avoidance as part of this field.

Europeans have a long tradition in fault-tolerant computing. The first triple-redundant computers were built in Czechoslovakia in 1956. At that time the reliability of the vacuum valves was not overwhelming, and the government incentives quite convincing.

Since then, European universities and research institutes hold a long record of activity. At first, as hardware was expensive, they emphasized coding techniques for error detection and correction. We even saw attempts to build a whole computer in two-rail, redundant logic.

With the coming of inexpensive processors, multiprocessor computers appeared as the royal way of fault-tolerant computing. Many thought that the failure of one processor in an array of 64 or 1,024 could easily be overcome by redistributing the work among the remaining ones. Unfortunately, this was easier said than done, and many multiprocessor projects that started with fault tolerance as a goal did not even achieve nonredundant operation. In one case, the computer only worked when some processors were down. Many remained purely parallel processing projects, although their names still spell the original goal.

Few universities maintain a fault-tolerant computer. Since hardware-implemented fault tolerance turns out to be painstaking work with little academic reward, it remained in the industrial domain. Researchers focus their interests on software-implemented fault tolerance in loosely coupled networks and databases. Software focus has the advantage that prototypes need not be demonstrated and also avoids catchy remarks from the audience. One should keep in mind that a triplicated computer fails at least three times more often than a simplex one.

The French have a practice-oriented, solid tradition in the field. At the IMAG laboratory of the University of Grenoble, a considerable number of projects led by Bernard Courtois relate to built-in self-test, microprocessor checking, and to failure mechanisms and redundancy at the wafer level. A nearby silicon foundry strongly motivates these projects.

The LAAS laboratory of Toulouse is world-known for its research on reliability modeling, software qualification, and fault-tolerant distributed systems. It has developed industrial networks and a dual-redundant multiprocessor for a satellite-based rescue beacon locator. The French National Council for Scientific Research, CNRS, owns the laboratory led by Alain Costes and Jean Claude Laprie, which works jointly with the nearby university and industry.

Not far from there, Airbus Industries builds the Airbus 320s. These are the first commercial aircraft controlled solely by a fault-tolerant, diverse computing system. Strangely enough, this development owes little to academia.

The Federal Republic of Germany also keeps very active. The German Mathematical and Data society, GMD, studies redundancy schemes for associative memories and databases with the team led by Karl-Erwin Grosspietsch. The Siemens Corporate Laboratories work on a completely self-checking processor. In the nearby Munich Technical University the ft(Unix) multiprocessor of F. Demmelmeir is based on off-the-shelf processors interconnected by a redundant bus.

The Goethe University in Frankfurt (Mario Dal Cin) tackles fault diagnosis through expert systems. At the Karlsruhe University, Klaus Echtle develops fault-masking protocols. The Asea Brown Boveri Research Center in Heidelberg developed a redundant programmable logic controller. The Dirmu (DIstributed Reconfigurable MUltiprocessor) of the University of Erlangen, built by Erik Maehle (now at the University of Paderborn) is one of the few prototypes that actually can be demonstrated.

Remember: A triplicated computer fails three times more often than a simplex one.

The interest group for fault-tolerant computing of the German Society for Informatics, GI, edits an appreciated newsletter. About half of the contributions are in English. You may receive it from the chairman, Ernst Schmitter. The interest group also organizes a conference on fault tolerance every second year. The 1987 GI conference at Bremerhaven attracted people from all over Europe and had very interesting contributions. In particular, conference managers organized the first exhibition of fault-tolerant computers.

The English, and especially the Center for Reliable Computing in Newcastle-Upon-Tyne with Brian Randell and Thomas Anderson, actively participate in the field of software recovery and reliable protocols. Their didactical contribution is one of the best references available today.

In Vienna the research group around Herman Kopetz works to develop a software environment for distributed, real-time computing, called Mars. Vienna hosted the last European IEEE conference on fault-tolerant computing, the very successful FTCS-16.

The Italians still try to mate parallelism and fault tolerance. The Polytechnic Institute of Milano (Roberto Negrini) studies a monstrous reconfigurable array for satellite imaging with some 20,000 processors. In Pisa, Luca Simoncini develops fault-recovery techniques for the MuTeam multiprocessor.

Finally, the Spanish also have been developing a multiprocessor at the University of Barcelona (Serrano) and work on reliability modeling at the Polytechnic University of Catalunya.

The European Community sponsors a technical committee for safety-related computing. Actually, it is the last group of EWICS (European Workshop on Industrial Computing Systems) that survived. The driving force behind EWICS is the safe use of computers in nuclear plants. Udo Voges of the German Nuclear Research Institute in Karlsruhe, W. Ehrensberger (Society for Reactor Safety), S. Bologna (ENEA, Italy), and W. J. Quirk (UK Atomic Energy Authority) are among the most active members.

The eastern countries also have activities in fault-tolerant computing. The 12th conference on fault-tolerant systems and diagnostics, FTSD-12, will take place this September in Prague. Jan Hlavicka of the Czech Technical University organizes it, and the topics are about the same as for FTCS.

Except for some projects directly financed by industry, none of these academic activities became an industrial product. One major reason is that fault tolerance cannot be dissociated from a particular application. General-purpose fault-tolerant systems are pills in search of a disease.

A few firms, mostly German, like Siemens, Paul Hildebrand, or Krupp-Atlas, offer general-purpose fault-tolerant computers. The customer still must do a lot to adapt these computers to its own process. In fact, the application departments of these firms are their main users. Recently, Siemens and Intel came up with a fault-tolerant multiprocessor workstation under the flag of BiiN. However, it is not yet clear for which market this workstation is intended. Architecturally, the workstation looks very much like Stratus Computer's Stratus/32 or Intel's defunct iAPX 432.

By contrast, many firms build ad-hoc fault-tolerant computers, like SAGEM (France) and MBB (Germany) for vehicle control, SEL (Germany) for railway control, or ABB (Switzerland), for burner or drive control. England (National Coal Board) and Germany (Frauenhofer Gesellschaft) built fault-tolerant networks for coal mines. The telecommunication societies, Ericcson (Sweden), Alcatel (France), and Italtel (Italy), have their proprietary designs for high-availability exchanges. Once you know for which purpose fault tolerance is introduced, you also can design for it.

Researchers in Europe face the same problem as elsewhere: Search for a unification theory did not leave the level of vocabulary definition. Theory tends to concentrate on local problems that are intellectually challenging but of little practical utility. Researchers comfort themselves in the importance of these topics by accepting each other's papers in journals and conferences, but actual implementations, manufacturers, and users are rare. At least, the researchers set the intellectual framework on which working solutions can be built. It is easier to find a solution once you know the problem.

Addresses

AFCET IMAG TIM-3
46, Avenue Felix Viallet
F-38031 Grenoble Cedex
France

Czech Technical University
Department of Computers
Prague, CSSR

EWICS TC7 Secretary
Mr. Rata J.M.A.
EDF
1, Avenue du General De Gaulle
F-92141 Clamart
France

German GI
Fachgruppe Fehlertolerierende Rechensysteme
Ernst Schmitter, Secretary
Siemens AG
ZT ZTI SYS1
Postfach 830955
8000 Munich 83
Federal Republic of Germany

Institute for Informatics IV
University of Karlsruhe
D 7500 Karlsruhe 1
Federal Republic of Germany

Institut fuer Praktische Informatik
Technical University of Vienna
Gushausstrasse 30
A-1040 Wien (Vienna)
Austria

LAAS-CNRS
7, Avenue du Colonel Roche
F-31077 Toulouse Cedex
France

Politecnico di Milano
Department of Electronics
Piazza L. Da Vinci
I-20133 Milano
Italy

Siemens Corporate Labs
Otto-Hahn-Ring 6
8000 Munich 83
Federal Republic of Germany

Technical University of Munich
Lehrstuhl fuer Prozessrechner
Frans-Josef-Strasse 38
8 Munich 40
Federal Republic of Germany

University of Newcastle
Center for Software Reliability
Claremont Tower
Newcastle-Upon-Tyne NE1 7RU
England

Wolfgang Johannes Goethe-Universitaet
Robert-Mayer-Strasse 11-15
D-6000 Frankfurt-am-Main
Federal Republic of Germany

Reader Interest Survey

Indicate your interest in this department by circling the appropriate number on the Reader Interest Card.

Low 174 **Medium** 175 **High** 176

Department

Micro Law

Richard H. Stern
Law Offices of Richard H. Stern
1300 19th Street NW, Suite 300
Washington, DC 20036

Framing prints, giving the Mona Lisa a moustache, speeding up video games, and marketing add-on software:

A comment on the Mirage case

A conflict is emerging in computer software copyright law over the legality of enhancement programs, which improve the performance of spreadsheets, database managers, and other business software. On one side, enhancers of such software desire to sell customers owning these software packages the enhancers' add-on programs that modify and improve the operation of the original software. Pitted against the enhancers are the original software sellers, who desire to control the identity of the computer programs in their customers' hands. Obviously, the sellers want to make any money to be had from selling enhancements to their programs, regardless of who conceives the enhancements.

To phrase the dispute in doctrinal terms, the original sellers may charge that the enhancers are preparing unauthorized "derivative-work" versions of the original computer programs in violation of the exclusive right of copyright owners to prepare derivative works. The enhancers respond by denying that there is any derivative work here; they also claim that the "first-sale" rule immunizes their enhancements. Sorting the problem out is complicated further by the Copyright Act's failure to define key terms involved in the definition of a derivative work.

Prior cases, which most of the time did not concern software-related copyrights, did not really test the limits of the doctrines involved. Nor did they lead to clearly outrageous results. The latest decision in this field is that of the Ninth Circuit Court of Appeals in *Mirage Editions, Inc. v. Albuquerque A.R.T. Co.* The *Mirage* case both tests doctrinal limits and leads to an unreasonable result. Although *Mirage* involves still another non-software case, it is a step in a direction that runs counter to important copyright policies relating to progress in software technology. These policies favor maximum encouragement of publication and public availability of technological innovations in software.

Consider the case of a program and its associated screen displays with a characteristic user interface and "look and feel" that are suboptimal in terms of esthetic quality or user friendliness. The program may also perform its tasks slowly, and may lack particular features or options that would make it more useful and attractive to users. At this point, suppose an entrepreneur enters the stage and writes an "add-on" program that improves or speeds up the original or provides supplemental features that the original seller failed to include. The add-on program may be assumed, by my hypothesis, to "work with" the original, so that customers necessarily buy a copy of each program from the respective sellers.

Do sellers of such add-on programs directly or contributorily infringe the copyrights of the original sellers, for example, their copyrights in screen displays? Is selling the enhancement in some way a copyright infringement? Or is the look and feel of the modified screen displays an infringement of the old screen displays? Does the seller cause the customer to commit a copyright infringement?

Until now, the conventional answer to these questions would be that there is no tenable theory of copyright infringement. The copyright owner's sale of a copy of the program to a distributor or consumer would give the copyright owner whatever reward for that particular copy to which it was legally entitled under the copyright laws. The copyright laws would give the copyright owner no power to exert further control over what the purchaser did with that copy. This principle is called the "exhaustion doctrine" or "first-sale rule."

To be sure, the purchaser could not lawfully make a different copy of the program or sell such a further copy. Apart from reproduction, however,

0272-1732/89/0400-0008$01.00 © 1989 IEEE

probably the purchaser would be considered free to repackage the program with another program or repackage it as part of a unit consisting of the program and a book explaining how to use it, or with hardware with which to use it. These acts would be protected by the exhaustion doctrine or first-sale rule.

Moreover, suppose that it were technically feasible to erase a fairly small number of portions, perhaps two percent or less, of the original diskette and replace them with different code. (Imagine that it is like a printed book in which correction tape could be placed over particular words, or that it is like an electronically addressable memory chip.) Or suppose that there is some free space on the diskette, so that the purchaser can add another program onto the diskette and that the added program would interact with the program already on the diskette. Probably—although this may go to the razor's edge of the principle—that too would be within the purchaser's privileges as owner of the goods and beneficiary of the exhaustion doctrine. This would be analogous to writing notes in the margin of a book and then reselling the marked-up book.

In the computer software industry, proprietors of application programs have grumbled at sellers of add-on software, but until a few months ago they have taken no legal action. Indeed, their threats have been derided, and add-on sellers have ignored them. Very recently, however, the seller of the leading database management program sued two sellers of add-on programs. The theory of the case is unclear, but the plaintiff may rely on the theory that derivative works are involved.[1]

In *Mirage,* a startling departure from what seemed settled law, the Ninth Circuit, the federal intermediate appellate court for the western US, found it to be copyright infringement for a purchaser of a book of art prints to mount individual prints from the book onto ceramic tiles and then resell them.

The plaintiffs in the *Mirage* case own copyrights in artworks, and a book containing artworks of a deceased artist, Nagel. Defendant Albuquerque A.R.T. is in the business of purchasing art prints and books containing prints, gluing the prints onto mounting material, and reselling the resulting product to the public. Albuquerque carried out its usual practice with Nagel prints that it took from purchased copies of the book, and the plaintiffs sued for copyright infringement. Upon cross-motions for summary judgment, the court found copyright infringement and enjoined Albuquerque from further similar acts with prints from the Nagel book.

The court acknowledged that Albuquerque did not "accomplish reproduction." What mattered, the court went on, was that Albuquerque prepared a derivative work and thus infringed the copyright. "What appellant has clearly done here is to make another version of Nagel's artworks . . . and that amounts to preparation of a derivative work" in violation of the copyright owner's exclusive right under section 106(2) of the Copyright Act to make derivative works.

Section 101 of the statute defines a derivative work as a work

> based on one or more preexisting works such as a fictionalization, motion picture version, sound recording, art reproduction, abridgement condensation, *or any other form in which a work may be recast, transformed, or adapted.* [Emphasis added.]

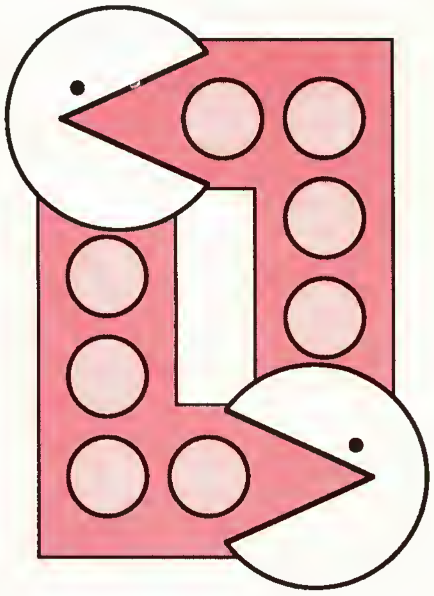
In the court's view, what Albuquerque did to the prints amounted to recasting, transforming, or adapting the work:

> By borrowing and mounting the preexisting, copyrighted individual art images without the consent of the copyright proprietors . . . [Albuquerque] has prepared a derivative work and [thus] infringed the subject copyrights.
>
> . . . Albuquerque has certainly recast or transformed the individual images by incorporating them into its tile-preparing process.[2]

Albuquerque sought to defend by asserting the exhaustion doctrine—that the sale of a copyrighted product by the copyright owner or its licensee provides whatever reward the copyright owner is entitled to. The particular product becomes no longer subject to the copyright owner's control. The court dismissed the defense mainly on the ground that the exhaustion doctrine does not apply in the case of the preparation of derivative works. The point has much to recommend it, as an abstract proposition. But whether the proposition is relevant to the facts of *Mirage* takes us right back to the basic question of what is the scope of the term *derivative work.*

The basic flaw in the court's approach is its concept of a derivative work. Sandwiching a picture between two sheets of plastic is not a preparation of a derivative work. A picture sandwiched between two sheets of plastic remains the same picture. It is not a derivative work; it is the same work.

The basic flaw in the court's approach is its concept of a derivative work.



With a few exceptions, past cases involving derivative works shed only limited light on whether a purchaser's modification of a copyrighted product will be considered an unauthorized and thus infringing derivative work. In one case, a defendant marketed an electronic kit containing a computer program that modified the Pac-Man video game when video-game arcade owners connected the kits inside their originally purchased Pac-Man units. The court did not have to consider whether the computer program that the defendant made and put in the kits was a derivative work based on the original computer program. Since the computer program in the defendant's kit took 90 percent of its code from the plaintiff's original Pac-Man program, it was clearly a copy and thus an infringing reproduction. A blatant copy does not invoke the issue of scope of a derivative work; the copy is simply an infringing reproduction.

On the other hand, the defendant's video-game audiovisual work, the screen displays that the defendant's computer program generated in the Pac-Man units that video-game arcade owners had purchased from the plaintiff, was not a derivative audiovisual work. The displays generated by the defendant's computer program did not resemble the original Pac-Man displays. The changes were so great that there was no copyright infringement, no appropriation of expression. Thus, it is not enough that one audiovisual work is derived from another when the two works are greatly dissimilar in appearance. Presumably, that means that if you paint a moustache and a Vandyke on the Mona Lisa, that is a derivative work. But if you paint over the Mona Lisa so thoroughly that she is unrecognizable (like a really thoroughly erased de Kooning), that is not a derivative work.

If you paint over the Mona Lisa so thoroughly that she is unrecognizable, that is not a derivative work.

On the assumption that copyright law protects screen displays, perhaps we can draw the conclusion from this case law that a great deal of change in a screen display will take it outside the scope of copyright protection (like the unrecognizable Mona Lisa), but a small amount of change may leave the defendant liable for preparing, or causing customer-users to prepare, a derivative work.

A very few cases reach a contrary result and condemn the defendant as an infringer for making a modification. The most interesting is another video-game case, *Midway Manufacturing Co. v. Artic International, Inc.*[3] This is probably the only reported decision that addresses the real economic issue in all of these derivative-work cases. The court of appeals' opinion indicates that the defendant sold a device that enabled owners of commercial video games to speed up the rate at which the video game was played, without otherwise changing the underlying computer program. The court found that the defendant's customers used the defendant's device to create unauthorized derivative-work versions of the audiovisual work, and that the defendant was therefore liable for contributory infringement.

The court acknowledged that speeding up a video game is much like playing a 33-rpm phonograph record at 45 or 72 rpm, so that the song comes through in a rapid falsetto. No infringement exists when a recording is speeded up, the court said, because there is no significant commercial gain from speeding up phonograph records, "so that record licensors would not care" if their licensees speeded them up. In effect, an implied license exists because of the presumed indifference of a copyright owner to economically valueless alterations.

But video games are different. Speeding up a commercial video game increases the revenue taken in. A speed-up kit therefore creates surplus value. Who should get it? To the Artic court, this was simple: "Video game copyright owners would undoubtedly like to lay their hands on some of that extra revenue and *therefore* it cannot be assumed that licensees are implicitly authorized to use speeded-up circuit boards in the machines plaintiff supplies." [Emphasis added.]

Now, in effect, there is no implied license, because the copyright owner is not indifferent to the revenue that could be derived from the new use. Indeed, the copyright owner is so far from indifferent as to sue the defendant for copyright infringement. The court concluded that creation of an unauthorized derivative work (and thus copyright infringement) occurs whenever the copyright owner could gain more than trivial additional value from separately marketing the second work, that is, the accused variation on the original work.

The *Mirage* and *Artic* cases illustrate broadened concepts of derivative works. In these cases, the court has expanded the scope of derivative work so much that the doctrine of derivative works collides with earlier cases in which, on similar facts, the courts held that the first-sale rule or exhaustion doctrine legitimated purchaser modifications of copyrighted products. Under the approaches taken in the *Artic* and *Mirage* cases, the sellers of add-on programs of the type discussed earlier could be held liable for copyright infringement for making and selling derivative works.

If an add-on program were sold by itself, rather than as part of a package including the original program, the principal question would become whether the seller causes its customers to make derivative-work versions of the original work, so that the seller is a contributory infringer. Enhanced screen displays can be regarded as derivative-work versions of the screen displays generated by the original computer program. Current US copyright law protects such screen displays, at least when the means for generating the screen display is fixed in a tangible medium such as a diskette.[4]

In addition, the interaction of the second computer program with the original computer program in the random-access memory of the customer's computer arguably causes a derivative-work version of the original program to exist in RAM. Or perhaps it causes a derivative-work version of the program's associated static displays or dynamic-display audiovisual works to be publicly displayed or performed by the computer when the two programs are executed together in a place of business. Thus, it is possible to maintain that the add-on program interacts with the original to cause a derivative work to be registered on the screen (in the case of screen displays) or in RAM (in the case of computer programs), or both.

In opposition to this theory of liability, the defendant add-on seller might retort that it did not sell copies of a

Continued on p. 85

Department

Micro Review

Richard Mateosian
2919 Forest Avenue
Berkeley, CA 94705-1310
(415) 540-7745

Abstraction seems to be the theme of this column. I didn't plan it that way, but the most interesting items that came my way this time dealt with the mathematical, logical, formal basis of computer science.

I hope that this is more than a random blip. I'm not saying that there isn't a need for books on how to program in or use C, Modula 2, Cobol 85, Glim, Genstat 5, Unix Document Formatting and Typesetting Tools, Windows 386, Hypertalk, Quick C, and the IBM PS/2. (Yes, these are all real books that occupy my "potential review" bookcase.) It's just that most how-to books don't repay the effort of reading them, because they don't seriously address the broader issues that students and active professionals must continually grapple with.

While I am certainly not a competent drama critic, I have also reviewed a play this time. It fits right into the abstraction theme.

Breaking the Code, Hugh Whitemore

Breaking the Code is a play based on Andrew Hodges' biography, *Alan Turing: The Enigma.* With Derek Jacobi as Alan Turing it became a surprise success in London in 1986-87. I saw it just recently at San Francisco's Magic Theatre, where it has now closed.

I'm sure every *IEEE Micro* reader knows that Alan Turing invented the Turing machine, a theoretical construct he used to show the limits of computability. Hugh Whitemore's view of this achievement gives insight into his play:

> Up to 50 years ago people thought there must be a structure into which you could slot any mathematical problem and come out with an answer. But Turing showed that this is wrong. Each problem requires fresh ideas and its own solution. And if there's no single way of telling right from wrong in math, which seems the most firmly decidable area of human knowledge there is, how can there possibly be an absolute right or wrong in morality?

The play's title, like the biography's title, alludes to Turing's work as a code breaker during World War II. Turing broke the German Enigma code, thereby, as he claims in the play, single-handedly winning the war. The play is a series of scenes, not arranged chronologically, taken from the periods of his education in the British Public schools, his wartime work, and his postwar professorship. It was during this latter period that the most notorious event of Turing's life occurred. He was jailed as a homosexual after stupidly telling a local constable about one of his affairs.

The Alan Turing who emerges from this play is emotionally childish and clumsy, intellectually sophisticated, and passionate about the mathematical concepts that occupy his mind. Sudden expository monologues and excerpts from a commencement address are woven smoothly into the story and serve to make visible to the audience Turing's passion for abstractions that are rarely the subject matter of plays.

Equations, Models, and Programs—A Mathematical Introduction to Computer Science, Thomas J. Myers

(Prentice Hall, Englewood Cliffs, N.J., 1988, 527 pp., $44)

This introductory computer science textbook uses an innovative approach. Most innovative approaches fail, and this one may not be the exception—only real classroom experience will tell. But I think that it's an approach worth trying, and Myers has done a remarkably good job of implementing it.

He begins with a 28-page introduction in which he states the essence of his approach, contrasts it with the traditional introductory course, persuasively sells his ideas to both the student and the instructor, and offers practical advice on how courses can be built around the book. The technical objective for the book is

> to prepare students for a program development process in which
>
> 1) Programs are designed as sets of modules, each with a visible specification and an invisible implementation.
> 2) Program specification is both precise and abstract, using formulas and diagrams as well as words.
> 3) Algorithms and data structures are designed from the specification.
> 4) Test cases are designed to reflect properties of the specification.
> 5) Debugging is based on assertions found in the specification.
> 6) Specifications and their implementations go through versions, with implementations altered frequently and independently, but specifications altered only with great caution after considering other modules which may be affected.

He claims that the traditional approach invariably fails to assign an appropriate level of importance to the specification. The usual introductory text, regardless of what its author says about the importance of specification, teaches the student to plunge into coding. Myers begins to teach programming techniques on page 336, although students can optionally write and run "equational programs" earlier in the course if certain programming languages, unknown to most of us (Scheme, Hope, ML, KRC), are available to them. The programming techniques are taught using Modula 2.

Several questions surely occur to you. What is he doing for 335 pages, and will computer science departments, or students for that matter, accept an introductory computer science course that involves no programming in the first semester? Myers anticipates and answers these questions in his introduction with a long parable about two academies of bridge building. The Hard Knock School of Practical Construction gives entering students hammers, nails, and planks and asks them to build small wooden bridges over ditches. The Look Before You Leap From The Ivory Tower School has its new students walk around on existing bridges, describe them, and work with plans for them.

What Myers is doing for 335 pages is the computer science equivalent of walking around on existing bridges and studying them. Students, who are assumed to have some facility in the symbolic manipulations of high-school algebra, are taught to compute with expressions describing the basic data objects and structures of computer science. This leads to formal proofs. The ability to build and manipulate models is developed, and in particular, state space search is given special attention.

Myers offers a compromise to the instructor or student whose fingers itch for the keyboard (Whatever happened to pencils and coding pads?): Finish the first three chapters (through page 212), then skip to page 336 and start to learn programming techniques. This concession, small as it seems, does allow programming to begin before the second semester.

What makes this offbeat approach to computer science have any hope of success—aside from its basic correctness—is the way Myers has done it. He continually engages the student in a dialogue. At any point in the text, a question-and-answer session may suddenly appear. Of course, Myers is putting words into the student's mouth, but many of these questions and answers ring true. Here's an example:

> We could just as well say that $x - y = z$ if $y + z = x$; subtraction is the inverse of addition.
>
> Q: That doesn't make sense. An inverse to producing z from x and y ought to produce x and y from z.
>
> A: True; it would be more precise to say that the $-x$ operation is the inverse of the $+x$ operation for any number x. For now, however, the important point is that when we used the 'look for an inverse' rule on addition and the 'look at repetitions' rule on 'Down,' we got to the same operation (subtraction). The inverse of repeated 'Up' is the repetition of inverted 'Up.'
>
> Q: I don't quite get that last remark; what's it for?
>
> A: . . . The reason I phrase it this way is that you will see statements of that general form again: 'The inverse of repeated F is the repetition of inverted F.' Remember it.

I could spend the entire column talking about this book, and I don't think it would be sufficient to let you decide whether to use it as a textbook or for independent study. If you're planning to teach an introductory course in computer science any time soon, you should get a copy of this book and give it serious consideration.

Abstract Data Types—Their Specification, Representation, and Use, Pete Thomas, Hugh Robinson, and Judy Emms (Oxford University Press, New York, 1988, 268 pp., $59.95 cloth, $26.95 paper)

Perhaps after my not-too-favorable review last August of a highly theoretical book that touched on the subject of abstract data types, Oxford decided that I needed something more down to earth. This book is about as down to earth as a book on abstract data types is likely to be. It is aimed at first- or second-year students, as an adjunct to an introductory course.

The authors set forth a systematic scheme for specifying abstract data types, beginning with their syntax, then specifying their semantics in one of two ways—the axiomatic and the constructive. The axiomatic approach is always needed somewhere along the way, but the constructive approach is an easy way to build complex data types from simpler, axiomatically defined ones.

The authors then try to implement examples in real programming languages, coming up immediately against serious problems. Encapsulation is the language capability needed to implement abstract data types. The authors explore how well encapsulation can be achieved in Pascal, Modula 2, and Ada. The answer, unfortunately, is that all of these languages (yes, even Ada) limit the degree to which the implementations of data types can be separated from their definitions.

The book is a genuine textbook. Checkpoints, with answers in the text, and exercises, with answers in the back of the book, form integral parts of the discourse. Unlike the other Oxford book reviewed in this column, this book is clearly written and carefully edited. I found few errors in it.

Logic with Prolog, Peter Gibbins (Oxford University Press, New York, 1988, 335 pp., $85 cloth, $32.50 paper)

Don't buy this book. In fact, don't even read this review unless you like purely negative reviews. The book is grossly overpriced, unbearably chatty, and extremely carelessly put together. Here are examples of the endless digressions and irrelevancies that I found so irksome:

> We, as programmers, tend to think of machine code as the fundamental representation of our computation, the one which really runs on the machine. But this is because we, as programmers, tend to lose interest in representations at lower levels than machine code, which is therefore the lowest level at which we are likely to find ourselves programming a machine. But there is nothing absolutely fundamental about machine code. Some processors can be programmed at the even lower level of microcode. A program written in the programming language Pascal, and the same program written in the machine language to which it compiles, are two different representations of the same computable function, the relation between them will be effected by a compiler, an object worthy of study by the computing scientist and one which may itself have been written in Pascal, or machine code.
>
> An engineer—civil, aeronautical, mechanical, electronic engineer—cannot afford to remain entirely ignorant of calculus. Calculus gives much of physics its mathematical basis and it also provides the engineer with the apparatus with which to apply the theories of physics to the business of making things work in the everyday world.
>
> 'Software engineering' is more than one of those fashionable phrases of which the discipline of computing is so full. It has the merit of carrying an important hidden message—that writing, designing, specifying computer programs, representations of the computable, must be treated as a form of engineering, an activity subject to the demand that its products have a guaranteed quality.

A competent editor would have red-penciled these and many more passages like them. This material might be interesting in a leisurely conversation over a pint of ale, but in an $85 book entitled

Logic with Prolog, we have a right to expect more wheat and less chaff.

Competent editing would also have eliminated the many typographical and technical errors. Over and over, things have one name at the beginning of a paragraph and another at the end. Or things become hopelessly jumbled, as when the author asserts twice, in two different forms that: "For every *y,* if *y* is not equal to *x,* then there exists an *x* such that *x* is less than *y.*"

Even when there is no technical error, the material is rarely presented crisply and cleanly. Again this is a failure of editing. The author states that he produced the book on a Macintosh with Writenow and printed it on a Laser-writer. So perhaps the (nominal) publisher simply shirked its editorial responsibility and placed the author's efforts at desktop publishing directly into print. If so, I can't see how they justify the price.

***Computing with Logic—Logic Programming with Prolog*,**
David Maier and David S. Warren
(Benjamin/Cummings, Menlo Park, Calif., 1988, 556 pp., $32.25)

Having whetted your appetite for Prolog and then having enjoined you from buying Gibbins' book, I felt that it was incumbent upon me to provide an alternative. This book is it. If you want to find out what logic programming is all about and get a good idea of what the implementation issues are, this is the book to read.

The book takes a slow approach, starting with Proplog, a language based on propositional logic. Propositional logic, and hence Proplog, is not a very powerful tool, but it serves to illustrate some of the basic ideas of logic programming and the design of an interpreter for a logic programming language.

Next, the authors introduce Datalog and present an interpreter for it. Datalog is a more powerful language than Proplog. It is based on predicate logic without function symbols.

After these two intermediate languages, the authors get to Prolog. Unfortunately, after all of this buildup, Prolog turns out not to be powerful enough either, so the authors then show how to add procedural extensions. Again, as with the Proplog and Datalog, the authors work out the implementation details, so the reader is left with a clear idea of what Prolog can do and how it does it.

Anniversary noted

I began writing this column in April 1987, so this is the start of my third year. As I've said many times in the past, please let me know what you think of these columns and tell me if there's anything you'd like to see reviewed. If you have a book, program, or super-computer that you'd like me to review, feel free to send it. Please don't send me hardware, especially complicated hardware, unless I can keep it for at least eight months. And while I won't damage it on purpose or sell it, I won't accept liability for anything that happens to it.

Keep those cards and letters coming.

Reader Interest Survey

Indicate your interest in this department by circling the appropriate number on the Reader Interest Card.

Low 177 **Medium** 178 **High** 179

FEATURE

The AT&T WE32200 Design Challenge

Here's how one design team boosted next-generation performance, retained compatibility, and beat the silicon clock.

Victor K. L. Huang
James W. Seery
William S. Wu
Saul K. Altabet
Michael J. Killian
Simeon Aymeloglu
Thaddeus J. Gabara
Aaron L. Fisher
Inseok S. Hwang
David W. Thompson

AT&T Bell Laboratories

Designers of high-end microprocessors face a major challenge: improving performance and versatility without compromising compatibility with previous-generation processors. AT&T met this challenge with its 32-bit WE32200 microprocessor. In fact, the WE32200 provides up to twice the performance of the WE32100.

Another element of our design challenge was to produce such a complex instruction-set computer in the shortest possible time. The silicon technology and circuit-design techniques within AT&T Bell Laboratories assisted our efforts. Sophisticated CAD tools and techniques for both circuit design and design verification also were essential. These advantages allowed us to achieve our objectives with a functional implementation of the first silicon chip.

During the design cycle, we saw that major reorganization of the chip architecture was necessary. The design strategy shifted from a purely technological upgrade to include architectural and feature enhancements. Based on architectural studies and user input, we introduced features to gain parity or performance without interfering with compatibility goals.

The 32200 can run at a frequency as high as 24 MHz and directly support multiuser/multitasking operating systems and high-level languages (HLLs). It employs thirty-two 32-bit, general-purpose registers to reduce the number of external memory accesses and facilitate the design of better optimizing compilers.

The 32200 also provides an expanded, orthogonal instruction set and additional addressing modes to improve the execution speed and efficiency of HLLs and the operating system. Support for packed, binary-coded, decimal (BCD) arithmetic speeds Cobol execution.

Instructions such as Call Process, Return Process, and User Call Process, by the same token, simplify and speed context switching for multiuser/multitasking operating systems. The 32200 provides loop-control instructions and addressing modes to improve string manipulation and table/array accessing.

The system also provides two I/O enhancements: arbitrary byte alignment and dynamic bus sizing. The first allows efficient access to unaligned databases. The second furnishes dynamic communications with both 16- and 32-bit ports.

See the accompanying box for an overview of the 32200 architecture.

0272-1732/89/0400-0014$01.00 © 1989 IEEE

Architectural Overview

The WE32200 consists of four basic sections: the main controller, the fetch unit, the execution unit, and the bus-interface unit (Figure A). The main controller contains a programmable logic array that decodes instructions and provides command sequences that other controllers use to execute instructions. This controller also processes the execution flow required for handling interrupts and exceptions.

The fetch unit consists of

- three PLAs,
- a 32-bit A bus,
- a 256-byte instruction cache,
- an 8-byte instruction queue, and
- a 32-bit arithmetic address unit (AAU).

The system prefetches instructions into the instruction queue to keep the pipeline full. It also places fetched instructions into the instruction cache. This process improves instruction-loop performance. A 32-word tag cache provides one tag entry for every two words in the instruction cache. The AAU performs 32-bit addition and accepts input from both the register array and the instruction stream, from which immediate and register offset values are derived.

The execution unit consists of

- four PLAs,
- a 32-bit A bus,
- a 32-bit C bus,
- two banks of sixteen 32-bit registers,
- a 32-bit ALU,

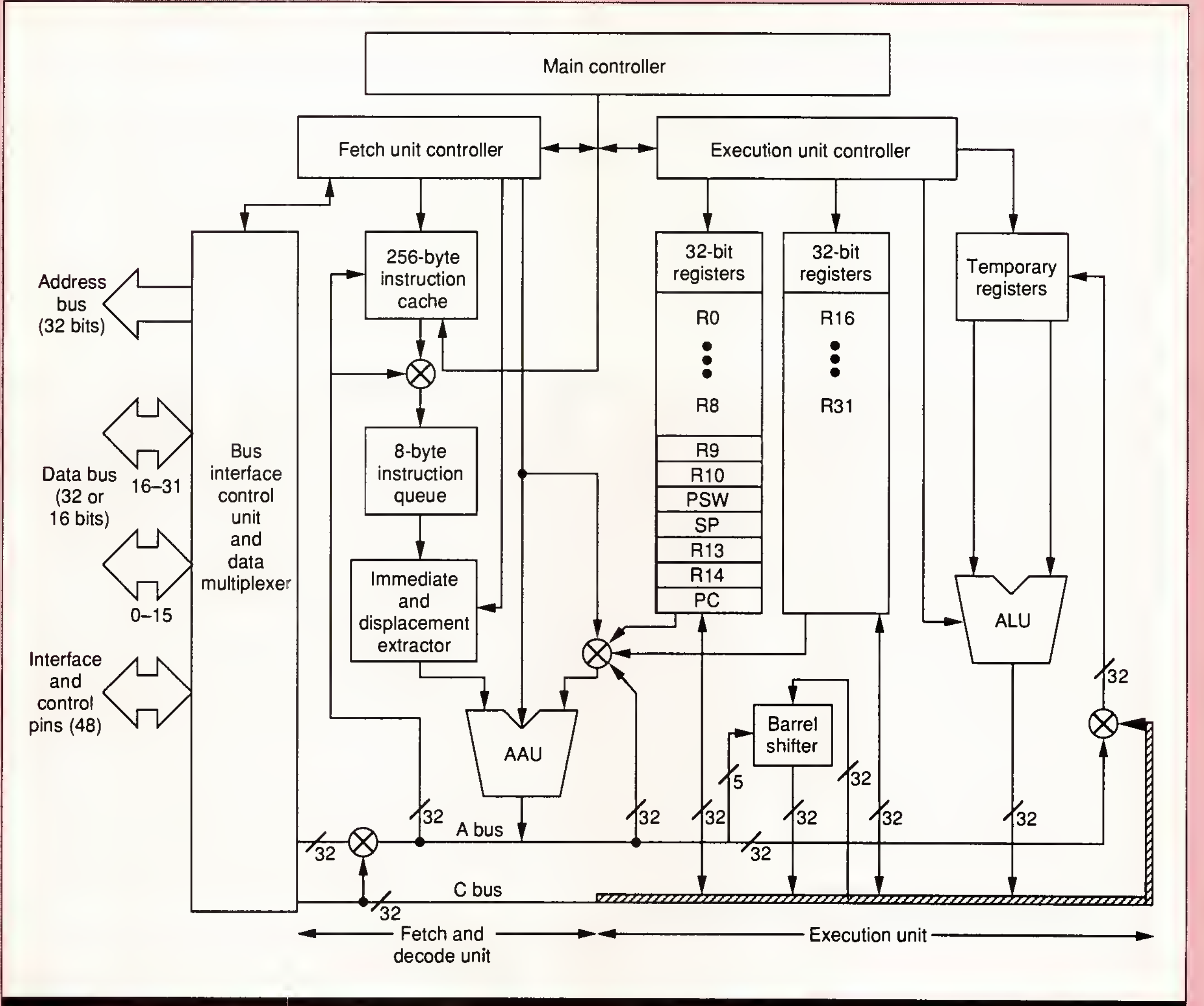


Figure A. WE32200 architecture.

Performance analysis

To achieve the performance goal, developers and users worked together to prepare a list of possible feature enhancements. We then estimated the performance impact of these enhancements by examining a number of benchmarks.

We first used *EDN*,[1] US-Steel Cobol, and C-language benchmarks chosen for their short runtimes and quick feedback. The *EDN* benchmark suite measures the performance of operations such as character-string search; bit test, set, and reset; linked-list insertion; and quicksort and bit-matrix transposition. The C benchmarks test recursive procedure calls, array and C-structure pointer operations, character-array operations, register- and external-variable usage, C-language switch statements and nested structures, and integer arithmetic. The Cobol suite measures the performance of operations such as display arithmetic (like BCD), character move/copy, and subscript move (array accessing).

We further narrowed the proposed enhancement list after examining these benchmarks. We implemented the remaining enhancements in a high-level architectural simulator so that more extensive benchmarks could be run. Some of these included the Berkeley benchmark suite (Ackermann's function, Puzzle, Search, and Sieve), as well as a modified *Byte* benchmark suite that consists of Fibo, Math, Ptr, Sieve, Sieve-i, Sieve-fi, String, and String_r. The test also included a benchmark suite containing the 10 most frequently used Unix commands for software development.

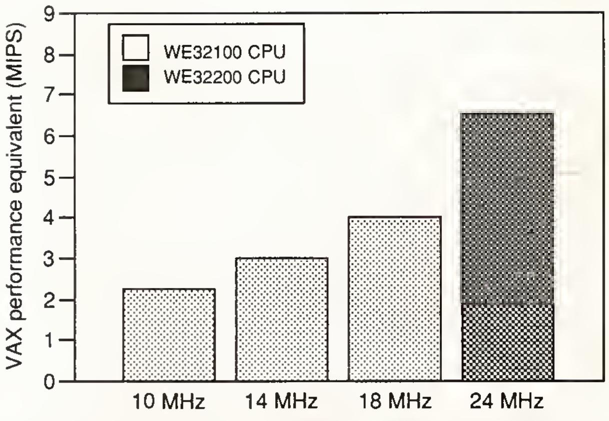


Figure 1. Berkeley benchmarks for the WE32100 and WE32200 versus the VAX 11/780.

We based the 32200's final architecture on these benchmark results. To determine if the proposed architecture met the design goals, we ran a subset of the benchmarks onto a cycle-accurate logic simulator. Once the needed performance increase was verified on the simulator, the design became silicon.

Figure 1 shows the performance of the 32100 and 32200 CPUs on Berkeley industrial benchmarks compared to a VAX 11/780 computer. Measurements are in

- a barrel shifter, and
- temporary registers.

The execution unit manipulates data into and out of the register arrays, operates the ALU, and transfers the data across the buses. The ALU performs logical and arithmetic operations, while the barrel shifter conducts rotate and shift operations. A PLA uses the ALU and the barrel shifter to perform multiplication and division.

The bus-interface unit consists of separate 32-bit address and data buses, and interface and control pins. These pins provide memory interface signals such as address and data strobes and include interrupt pins for 16 levels of interrupts, and hardware/software development pins. This unit also contains input and output data multiplexers to align or unalign data on the fly. It provides arbitrary byte alignment and interfaces to 16-bit ports.

The chip operates with a four-phase clocking scheme. Two clock signals—which are 90 degrees out of phase—are input to the chip, buffered, and distributed through the chip. Local clock decoders generate some or all of the four clock phases as well as other timing signals.

The left side of the data-path section is part of the instruction fetch and decode unit. The A bus provides the primary communications path for components within the fetch-decode unit. The right side of the data path is part of the execution unit. The C bus provides the communications link for portions of the execution unit.

One of the major challenges in designing the data path was providing for high-speed data transfer between the A and C buses. In the 32100, a bidirectional set of drivers between the two buses handled this transfer. Because this technique required charging two buses, it was too slow for the 32200. We eliminated this communications path to increase 32200 clock speed. Instead, the design provides that all functional blocks directly access relevant buses through local multiplexers.

terms of million instructions per second. Note that a benchmark generally measures the raw performance of a specific subset of the instructions/functions supported by a microprocessor. A benchmark *does not* measure the performance of the design ease of a microprocessor at high frequency[2] or of a realistic multiuser system. The only truly effective benchmarking methodology is a one-to-one comparison of two actual systems running the same application.

Functional partitioning

Because the 32200 is designed for high-end multitasking/multiuser applications, its functional partitioning differs from many of its contemporaries. To reduce chip cost, many 32-bit microprocessors incorporate system-level functions such as memory management and math support. However, this strategy reduces the silicon available for each function.

Because our primary goal was high performance—rather than low cost—we decided to implement the CPU, memory management unit (MMU), and math acceleration unit (MAU) separately (Figure 2). This approach furnished the silicon necessary to implement a CPU with thirty-two 32-bit registers, a 256-byte cache, and a large instruction set. The MMU cache could combine a 64-entry, fully associative, page-descriptor cache with a 4-Kbyte, two-way, set-associative data cache. A 3.5-million-Whetstone math coprocessor provides the hardware support needed for transcendental functions.

Expanded register set

Because the 32000 family was originally designed as a Unix engine for AT&T's 3B minicomputers, developers optimized its architecture and instruction set to execute HLLs and multitasking operating systems. The 32100 provided sixteen 32-bit registers (R0-R15). The 32200 extends this strategy by adding sixteen 32-bit registers[3,4] to take advantage of continuing advances in compiler technology and to improve the execution speed of HLLs. The additional registers give compilers a place to store frequently used data, thereby reducing CPU-to-memory and execution times. Extensive studies of the compiler resulted in the allocation of eight registers for users and another eight for the operating system.

This organization produces a total of 20 user and 11 kernel registers. Table 1 on the next page shows the programmer's model. The 20 user registers (R0-R10, R12, and R16-R23) are readable and writable in any execution mode. Since R16-R23 are intended for global variables and temporaries, the instructions that are used to save and restore registers across procedure calls were not extended for these new registers. Nevertheless, Push/Pop instructions can save and restore these registers from the stack. However, since R16-R23 belong to the new process model, they are saved and restored when a process-switch operation (user-saving process) occurs. The 11 privileged registers (R11, R13, R14, and R24-R31) are readable in any mode, but are writable only in kernel mode. The eight privileged registers

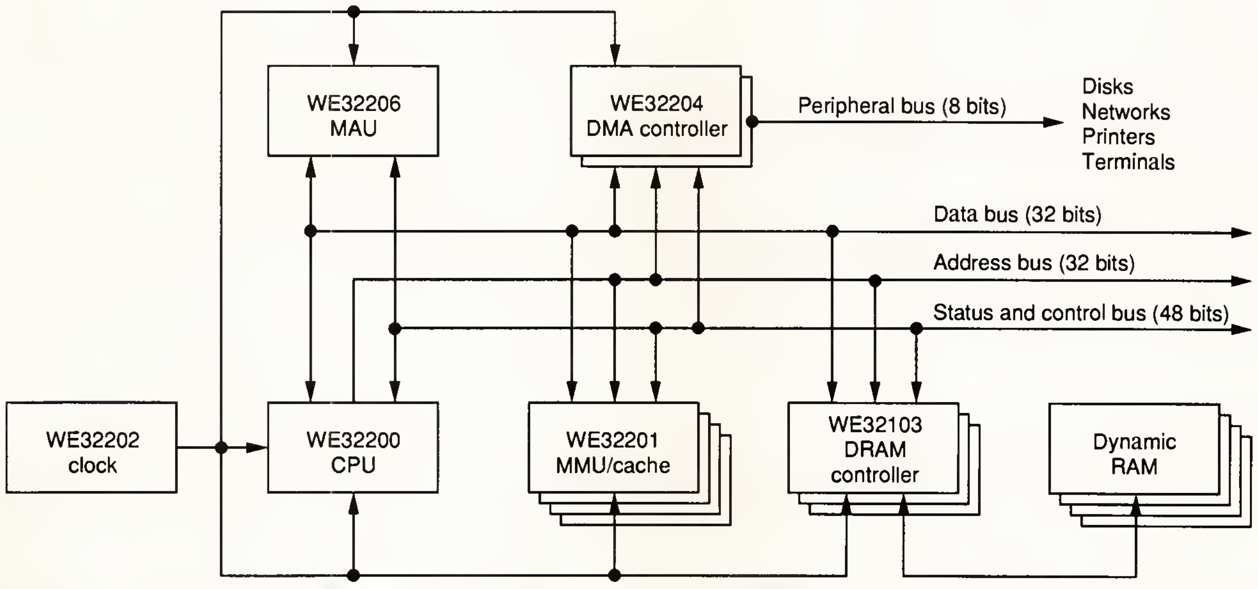


Figure 2. The WE32200 chip set includes a 32-bit microprocessor (WE32200), an MMU/data cache (WE 32201), a 32-bit DMA controller (WE32204), and a 3.5-million-Whetstone math coprocessor (WE32206).

Table 1.
Programmers' model of WE32200 registers.

Register number	Type
0-8	General purpose
9	Frame pointer
10	Argument pointer
11	Process status word, privileged
12	Stack pointer
13	Process control block pointer, privileged
14	Interrupt stack pointer, privileged
15	Program counter
16-23	General purpose
24-31	Privileged

Table 2.
Additional instructions for the WE32200.

Mnemonic	Instruction function
ADDPB2	Add packed BCD dyadic
ADDPB3	Add packed BCD triadic
SUBPB2	Subtract packed BCD dyadic
SUBPB3	Subtract packed BCD triadic
SETX	Set X bit in PSW
CLRX	Clear X bit in PSW
PACKB	Pack BCD
UNPACKB	Unpack BCD
UCALLPS	User call process
RETQINT	Return from quick interrupt
DTB	Decrement and test byte
DTH	Decrement and test halfword
TEDTB	Test equal, decrement and test byte
TGEDTB	Test greater or equal, decrement and test byte
TGDTB	Test greater, decrement and test byte
TNEDTB	Test not equal, decrement and test byte
TEDTH	Test equal, decrement and test halfword
TGEDTH	Test greater or equal, decrement and test halfword
TGDTH	Test greater, decrement and test halfword
TNEDTH	Test not equal, decrement and test halfword
CASWI	Compare and swap word interlock

(R24-R31) added to the 32100 design can be allocated for time-critical applications that do not need the full register set. They can also be used to improve context save and restore time. Since R24-R31 do not belong to a process in the usual sense, they are not generally saved across a process switch (see the section on support for operating systems).

Instruction-set orthogonality remains the same for all new registers, which work with any 32100 or 32200 addressing mode. The 32200 supports two process models for the new registers. The process-handling instructions can save and restore the 32100's 12 user registers, as well as the 32200's eight additional user registers.

Extended instruction set

In addition to providing a larger register set, the 32200 can support HLLs by providing an expanded-yet-orthogonal instruction set. In many microprocessors, several instructions use only a portion of the registers and addressing modes. This situation forces the writer of compiler code to handle a number of special cases, which complicates compiler development. Virtually all 32200 instructions use all registers and addressing modes.

The 32200 instruction-set object code is compatible with the 32100. It supports five data types: bytes, halfwords, words, bit fields, and BCDs. The CPU interprets bytes, halfwords, and words as either signed or unsigned in arithmetic or logical operations. Multiple data types can also mix within the same instruction. Program-control instructions such as Branch, Jump, and Return alter the sequence of program execution for flexibility.

As previously mentioned, the 32200 employs packed BCD arithmetic to provide stronger support for Cobol. Because the dual-radix arithmetic logic unit (ALU) directly performs BCD addition or subtraction, the 32200 can execute decimal-oriented languages such as Cobol faster than architectures that need an extra adjustment instruction after every BCD addition or subtraction. The ALU performs the binary addition. A BCD correction circuit at the ALU's output converts the result back into BCD format.

The 32200 also provides support instructions for BCD functions. SETX and CLRX set and clear the ALU's carry in/out bit. PACK and UNPACK convert data between packed and unpacked decimal formats. This instruction set enables the 32200 to execute Cobol up to 15 percent faster than the 32100. (These measurements result from US-Steel benchmarks and do not include clock-rate increases.)

To speed string manipulation and table/array access, the 32200 provides 10 loop-control instructions (see Table 2). These instructions—which are generally placed at the end of a loop—test condition flags, decrement a counter, and compare the counter's new value with the number 0. A flag-comparison step tests for multiple flag settings. Following the flag comparison, the CPU can perform either a byte- or halfword-displacement branch. For string/character-type applications like *EDN* benchmark E, as much as a 50-percent-performance improvement (not including clock-rate increase) can occur in conjunction with certain addressing-mode uses (described later).

An added Compare and Swap Word Interlock instruction (CASWI) speeds and simplifies communication between processors in a multiprocessor system. It combines a compare operation with an indivisible read-modify-write operation to reduce the amount of code required to provide security for shared resources. The fact that the read and write (if required) perform under an interlocked status code and cannot be interrupted guarantees security. The CASWI instruction also provides user-level semaphore operations.

Operating-system support

Both the 32100 and 32200 have several instructions that directly support multitasking processor-oriented operating systems. We did not build an operating system into the CPUs. However, a particular process model is implicit in the 32100's architecture.[5] As shown in Figure 3a, the 32100's process control block (PCB)—an image of its process model—contains a a copy of the processor resources that each process uses.

In addition to supporting the 32100's process model, the 32200 uses another process model that includes additional registers. Figure 3b shows the 32200's PCB. In both CPUs, the same instructions or primitives save and restore a copy of a process. The 32200 supports the old and the new process models via a compatibility bit in the process status word (PSW).

The 32100 and 32200 each provide two mechanisms for manipulating processes, controlling transfers to the operating systems, responding to interrupts, and handling exceptions.

A *process-switch mechanism* provides interrupt and exception handling in addition to process switching. Two privileged primitives—Call Process and Return-to-Process instructions—furnish the operating system with process switching. A process-switch operation saves the context of the executing process into its PCB and loads the context for the next process from its PCB into the corresponding set of registers.

A *controlled-transfer mechanism* controls entry into a procedure or handler and the replacement of the PSW. Some operating systems use this mechanism for system calls. Each CPU provides two primitives, Controlled Call and Controlled Return. Here, the Controlled Call instruction has two operands that serve as a double-table index for determining the new PSW and the appropriate branch address. The system pops the pro-

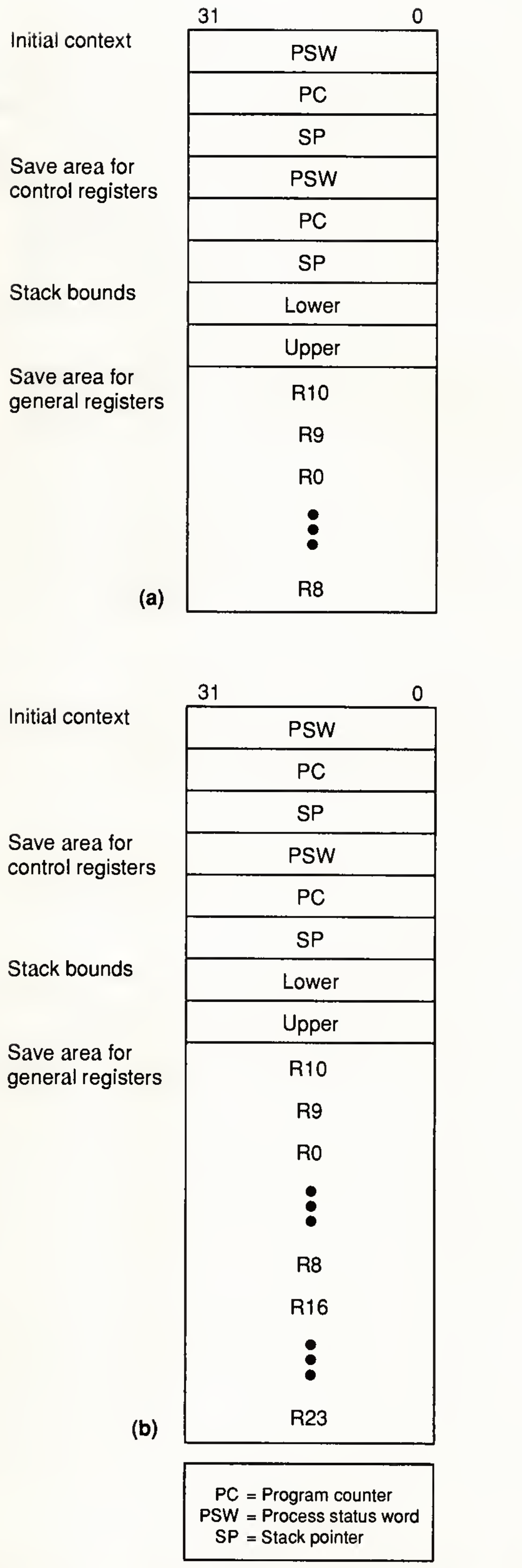


Figure 3. The process control blocks for the WE32100 (a) and the WE32200 (b) contain register values.

Table 3.
Addressing formats for the WE32200.

Mnemonic	Addressing modes
+(%orn)	Automatic preincrement
−(%orn)	Automatic predecrement
(%orn)+	Automatic postincrement
(%orn)−	Automatic postdecrement
expr(%ory,%orm)	Indexed with byte/halfword displacement
%orm[%orn]	Index with scaling
%orm	Register of %or16-%or31
(%orm)	Register deferred of %or16-%o31
expr(%orm)	Byte/halfword/word displacement of registers %or16-%or31
*expr(%orm)	Byte/halfword/word displacement deferred of registers %or16-%or31

%orn	Register %or0-%or31
%ory	Register %or0-%or15
%orm	Register %or16-%or31
expr	An expression that yields a data value

gram counter (PC)/PSW pair off the stack on a Controlled Return.

For some operating systems, it makes sense from programming and performance standpoints to have a single controlled entry directly perform a process switch. The 32200 provides this feature via an additional Controlled Call mechanism/instruction. Similar to the Privileged Call process instruction in the 32100, UCALLPS saves the context of the previous process and switches in the context of the next process. Instead of using an entry table, an implicit address that contains the PCB location of the new process serves as a single entry point.

All 32200 instructions are restartable. The system can back up and restart a faulted instruction as if the fault had not occurred. If a fault occurs during instruction execution, the CPU saves the PC for the faulted instruction onto the stack. The destination registers are not updated when the fault occurs. When the fault (for example, a page fault) is corrected, the instruction can reexecute.

Addressing modes

To further simplify assembly-language programming and improve efficiency and execution speed, the 32200 provides several new addressing modes. The 32100 provided the following addressing modes:

- absolute and absolute-deferred;
- displacement and displacement-deferred; and
- immediate, register, and register-deferred.

It also provided an expanded-operand addressing mode that performed data-type conversions.

In addition, the 32200 supports automatic preincrement, predecrement, postincrement, postdecrement, index, and scaled-index modes (see Table 3). The indexed register with byte- and halfword-displacement modes—along with the indexed register with a scaling mode—improves array-manipulation performance. This approach speeds operations such as changing encodings, removing multiple blanks, and removing the extraneous characters that are heavily used in compilers and in Cobol. The indexed register adds the contents of two registers, one from each register bank. Because the register banks are separate, parallel register reads can occur.

The automatic preincrement, predecrement, postincrement, and postdecrement addressing modes speed the manipulation of multiple stacks in artificial-intelligence languages. These modes provide register pointers to memory, which stores the values used to increment or decrement the register's address. The pointer to memory automatically moves to the next piece of data each time it is used. The value can be 1, 2, or 4, depending on the data size.

These modes presented an interesting design challenge. An incomplete instruction modifies the register content before a memory-location access. If the memory access faults, the instruction does not restart.

To overcome this problem, designers added a restartability section to the 32200 to keep track of the registers that such modes modify. When a fault occurs—and before it enters the fault handler—this section restores the original values to the register. To do this, the section has to keep track of

- how many instructions inhabit the pipeline,
- how many automatic addressing modes comprise each instruction,
- which registers change because of these modes,
- which increment—1, 2, or 4—modified the automatic-mode register, and
- which instruction has a fault.

Two instructions can use up to seven registers with automatic addressing modes. The restartability section can fix all seven registers when a fault occurs. If the faulted instruction appears first in the pipeline, all the automatic-mode registers need repair. If the second instruction faults, then the automatic-mode registers that are solely defined by the second instruction need repair.

Arbitrary byte alignment

Apart from its internal architecture, the 32200 provides a number of I/O features. Arbitrary byte alignment allows the 32200 to efficiently access databases in which the data crosses a word boundary. A CPU without ABA issues a fault when a user program specifies an address in which different bytes lie in different memory words.

When a CPU with ABA detects an access that crosses a word boundary, it decomposes the address into the most efficient halfword/byte combinations that don't cross word boundaries. The CPU may have to perform more than one access to fetch an address, but it avoids the overhead of a fault and the subsequent operating system intervention.

Figure 4 illustrates how a 32200-based system implements ABA. The upper 30 bits of the address bus select the word in which the data resides. The lower 2 bits, in conjunction with the data size (from 4 bytes to 0 bytes), select the bytes within the word. For example, if the CPU wants to write data AB of the data-size halfword (2 bytes) to address X10—where X is the upper 30 bits of the address and 10 is the lower 2 bits—the data will write to memory as shown in Figure 4.

Figure 4 also shows how the 32200 handles an address that crosses a word boundary. For example, if a user program specifies that the data word Y01 be read, the CPU

- generates an address of Y01 with a 3-byte data size,
- receives data CDE,
- generates an address of Z00 with a 1-byte data size,
- receives data F, and
- reorders the data bytes to form the correct data word.

Designers defined a probe-type access to provide ABA and maintain restartability. A probe data transfer is a write access of 0-byte data size. Because no bytes activate during a probe access, no data writes to memory.

A probe access is needed when a CPU in virtual mode uses demand paging and performs a read or write access that crosses a page boundary. Figure 5 illustrates what

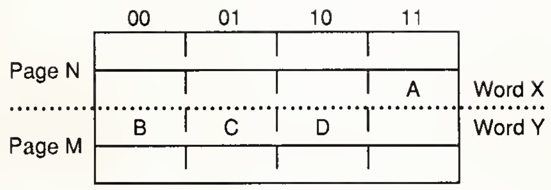
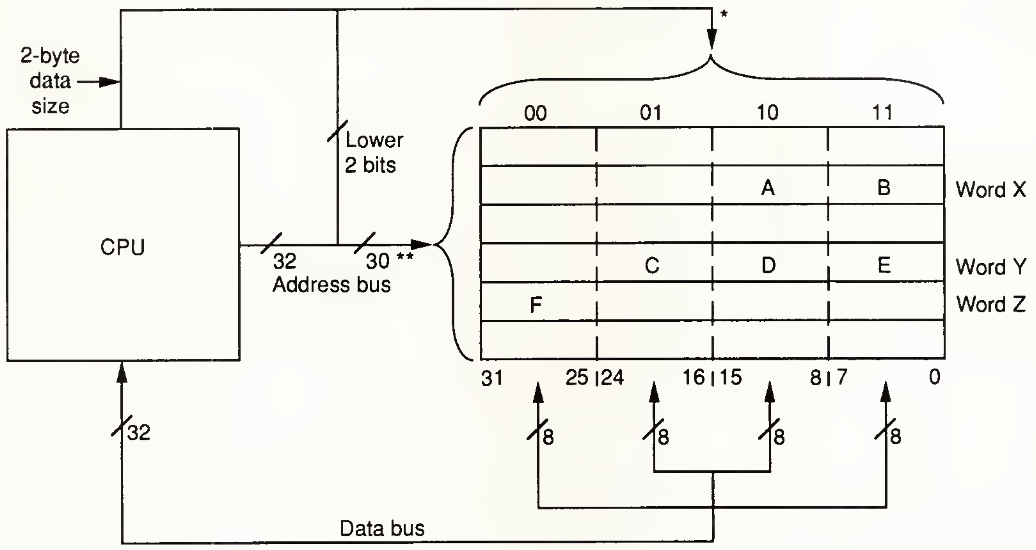


Figure 5. An example of arbitrary byte alignment by the alignment across page boundaries.



* The lower 2 bits of the address bus and the data size select the bytes within the word.
** The upper 30 bits of the address bus select the row of the RAM.

Figure 4. Arbitrary-byte-alignment organization.

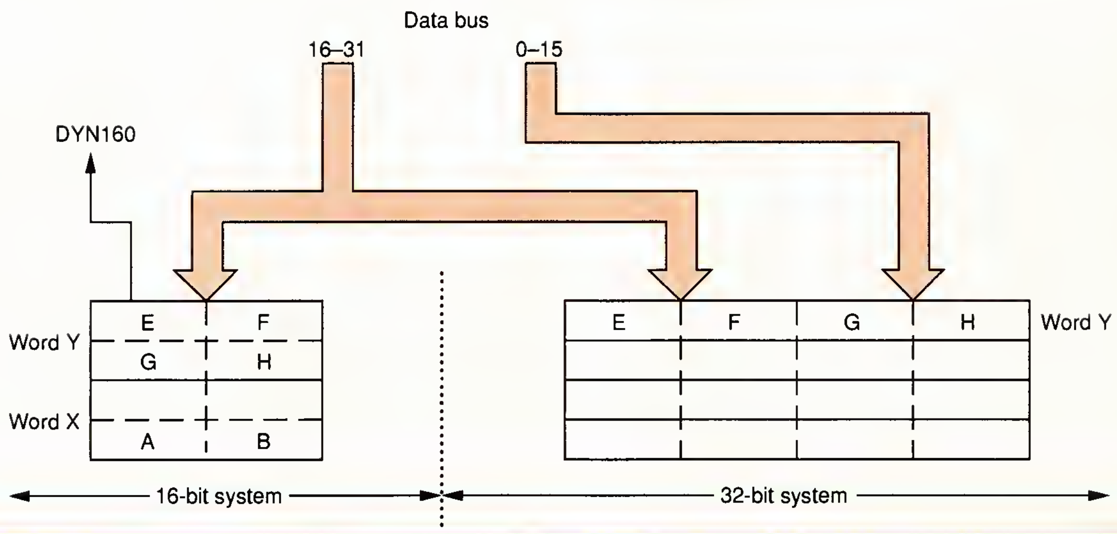


Figure 6. Data-bus connection for 16- and 32-bit ports.

happens if the CPU doesn't have a probe data type. Suppose the CPU wants to write the word ABCD to X11. Without a probe data type, the CPU tries to write A to page N and BCD to page M. However, if page M isn't resident in main memory, the write of BCD faults. Since the write of A to page N has already modified main memory, the instruction won't be restartable.

On the other hand, a CPU with a probe data type issues a probe transfer on the first access to location X00 on page N. On the second and third accesses, it writes BCD to page M and A to page N. Because no data was written on the first access, the instruction can restart whether the fault occurs on the first or second access.

Dynamic bus sizing

This I/O feature allows dynamic communication with 16- and 32-bit ports. The key word here is *dynamic*. During a memory access, the CPU doesn't know whether it is addressing a 16- or 32-bit port until it receives an acknowledgment from that port. If it receives the 16-bit port's acknowledgment (DYN160), it generates another access to deliver any untransferred data.

Figure 6 shows the data-bus connection for the 16- and 32-bit ports. It also shows a write operation in which the word EFGH is written to location Y. If the port is only 16 bits wide, it breaks the 32-bit word into two 16-bit words after it receives DYN160. It then writes EF to location Y00 and GH to location Y10.

A potential problem arises if the CPU wants to read or write a halfword to a 16-bit port. Let's say the CPU wants to write the halfword AB to location X10 via a 16-bit port. Since the CPU always assumes that it is communicating with a 32-bit port, it uses data-bus lines 0-15 to write the data (Figure 6). The problem is that by definition the 16-bit port connects to data-bus lines 16-31. Consequently, it doesn't see the data that the CPU writes.

To solve this problem, the 32200 provides *byte replication*. The CPU sends a duplicate of the 16-bit word on lines 0-15 as well as on 16-31. The CPU uses the same strategy for writing bytes of data. When it writes a byte to memory, it duplicates the byte in all four byte locations within the data bus. DBS and ABA work together on the 32200. Programs execute without a fault in a system that uses ABA with a 16-bit port. However, performance can suffer because memory accesses break into as many as four smaller units.

Circuits

The 32200 fabrication uses a 1.25-micrometer, twin-tub, complementary metal-oxide semiconductor technology in its fourth generation at AT&T. The general-purpose process takes 5 volts. This single-level-metal, silicon-gate technology self-aligns with low-resistivity tantalum silicide at its lowest level interconnection. See the accompanying box for circuit-layout information.

The devices are fabricated on P+ substrates with a P-epitaxial layer, which makes them immune to latching up. A lightly doped drain structure improves the stability of the N-channel transistors. This structure, in turn, allows the design of narrow N-channel devices without any hot electron effects, even at submicrometer channel lengths. The nine-mask (including passivation) process features minimum effective channel lengths as low as 0.75 micrometers.[6] It also provides layout rules with a minimum metal and silicide pitch of 2.5 micrometers.

We used Domino CMOS,[7] multiphase dynamic-logic,[8] and pseudo-N-channel-MOS circuit-design

Circuit Layout

We used more than one layout style because a multimethodology approach improves design efficiency, decreases overall chip area, and achieves the desired circuit performance. Any particular style used in a given circuit section depends on the requirements for that section, such as area, performance or design efficiency, and flexibility.

Full custom layout at a geometric level occurred in sections that needed absolute minimization of area, such as the chip I/O frame and instruction cache.

We selected gate-matrix techniques[9] for sections in which pitchmatching, abutment, and performance were important. The main involved area was the CPU data path. This technique also possesses the advantage of technology upgradability. Since the 32200 is upwardly compatible with previous 32100 designs that also use gate matrix, many data-path elements were reused via recompiling their symbolic layout descriptions into the 1.25-micrometer technology. This recompilation resulted in schedule reductions and a faster route to a manufactured product.

To these ends, we performed many design activities in parallel (architecture, logic, circuit, and layout). This approach, however, inherently introduces changes to designs that are already complete. This especially affected the control sections of the processor that comprise 50 percent of the total chip area. To allow flexible modification of this circuitry late in the design cycle, we extensively applied automatically generated PLAs coupled with standard cell circuitry.

The remaining control section, I/O control, benefitted from a modified standard cell methodology. Here, fixed-height custom blocks were designed to contain critical timing paths within them. This design minimized the impact of using automatic-routing software on the performance of this very critical section. This technique also allowed quick reimplementation of the section to change its aspect ratio.

technologies to provide high performance per chip area. The highest performance sections employed Domino CMOS, which yields propagation delays that are much lower than those attained in static implementations. These sections include the ALU, the AAU, and the barrel shifter.

Because it suits regular layout structures so well, we used multiphase dynamic logic extensively to implement the CPU's control section. This section is based primarily on programmable logic arrays. Since PLA outputs control all CPU operations, we maximized PLA operating frequencies and took care to ensure that these signals would arrive at their destinations on time.

Fabrication employed psuedo-NMOS circuitry in CPU regions that needed static circuitry but could not use standard, fully complementary CMOS gates because they were either too slow or consumed too much area. Taking advantage of resistively biased P-channel-MOS transistors as pull-up devices in the logic tree achieved static-circuit operation without consuming as much area.

Timing and analysis

Achieving a 24-MHz frequency for the 32200 was not just a matter of upgrading technology from 1.75- to 1.25-micrometer design rules. Even a 4-MHz increase in base operating frequency demanded redesigning or reimplementing several structures based on the 32100. This challenge nearly equalled that of upgrading 32100 functionality. Some major affected structures were

- the internal instruction bus;
- the AAU;
- the instruction-cache, hit-detection circuitry;
- the A-bus and C-bus multiplexers; and
- three major control PLAs.

Design tools helped identify and enhance 32200 timing improvements. A PLA timing simulator enabled designers to increase operating frequency by identifying and overcoming excessive loading on input and output lines. For example, the data led us to reduce the loading on output lines by recombining the lines outside of the PLA.

Another design tool—which is independent of test coverage—automatically examined and analyzed timing for all logic paths. This tool allowed hundreds of faulty paths to be identified and corrected prior to silicon fabrication. These corrections ranged from transistor tuning to logic redesign, all of which we rechecked after running the path-analysis tool a number of times.

By taking advantage of these design tools, and employing clever architectural and circuit-design techniques, we achieved full functionality in the first silicon-chip implementation. Given the imposing design goal of an up-to-two-times performance increase over the previous-generation 32100—and extensive architectural extensions—achieving a successful first silicon implementation is all the more amazing. In summary, the 32200 was a technological success. ▣

Acknowledgments

The authors thank the members of the VLSI Processor Development Department for the effort and dedication that made it possible to achieve a fully functional, first-silicon implementation in such a short time.

References

1. T.C. Cooper et al., "A Benchmark Comparison on 32-bit Microprocessors," *IEEE Micro*, Vol. 6, No. 4, Aug. 1986, pp. 53-58.
2. A.G. Martin and N.R. Miller, "Three 32-bit Chips Boost Multiuser Performance," *Computer Design*, Nov. 27, 1986.
3. A. Lunde, "Empirical Evaluation of Some Features of Instruction Set Processor Architectures," *Comm. ACM*, Vol. 20, No. 3, Mar. 1977, pp. 143-153.
4. C.A. Wiecek, "A Case Study of VAX-11 Instruction Set Usage for Compiler Execution," *Proc. Symp. Architectural Support for Programming Languages and Operating Systems*, ACM, New York, Mar. 1982, pp. 177-184.
5. A.D. Berenbaum et al., "The Operating System and Language Support Features of the BELLMAC-32 Microprocessor," *Proc. Symp. Architectural Support for Programming Languages and Operating Systems*, ACM, New York, Mar. 1982, pp. 30-38.
6. K.H. Lee et al., "Lightly Doped Drain Structure for Advanced CMOS (Twin-Tub IV)," *Proc. Int'l Electron Devices Meeting*, IEEE Electron Devices Society, Dec. 1985, pp. 242-245.
7. R.H. Krambeck et al., "High Speed Compact Circuits with CMOS," *IEEE J. Solid State Circuits*, Vol. SC-17, June 1982, pp. 614-619.
8. H.F.S. Law and M. Shoji, "PLA Design for the Bellmac-32A Microprocessor," *Proc. IEEE Int'l Conf. Circuits and Computers*, IEEE CS Press, Los Alamitos, Calif. (microfiche), Sept. 1982, pp. 161-164.
9. A.D. Lopez and H.F.S. Law, "A Dense Gate Matrix Layout Method for MOS VLSI," *IEEE Trans. Electron Devices*, Vol. ED-27, 1980, pp. 1,671-1,675.

Victor K. L. Huang supervised the design and development of the WE32200 CPU. His responsibilities at AT&T Bell Laboratories also included the design of AT&T's 4-, 8-, and 32-bit microprocessors. Huang received his BSEE from the Virginia Military Institute and his MSEE and PhD from the University of Virginia. He is a senior member of the IEEE and a member of the IEEE Computer Society, the Industrial Electronic Society, and the editorial board of *IEEE Micro*.

James W. Seery was the lead design engineer for the WE32200 CPU. Before coming to AT&T in 1981, he worked at Lockheed Electronics. Seery received his BSEE and MS in computer science in the same year and a professional degree in electrical engineering from Stevens Institute of Technology. He is a member of Tau Beta Pi.

William S. Wu was the architect for the WE32200, as well as a member of the design team. His interests include very large scale integration architecture, interconnection networks, and multiprocessor architecture. Wu received his BSEE from the University of Minnesota, his MSEE from Carnegie Mellon University, and his PhD from the University of Michigan. He is a member of the ACM, Eta Kappa Nu, Tau Beta Pi, and the IEEE Computer Society.

Saul K. Altabet is a Member of the Technical Staff. As part of the WE32200 design team, his primary responsibilities were the architectural and logic design of the WE32200's I/O. Prior to joining AT&T, he worked on restructural VLSI at MIT Lincoln Laboratories. Altabet received a BS in electrical engineering and chemistry and an MSEE from the State University of New York at Stony Brook. He is a member of Phi Beta Kappa.

Michael J. Killian is the supervisor of the VLSI Processor I group in AT&T's Data Systems Division. He is responsible for the design and development of 32-bit VLSI microprocessors such as the WE32100 and WE32200. He holds BSEE and MSEE degrees from the State University of New York at Stony Brook. Killian is a member of the IEEE Computer Society.

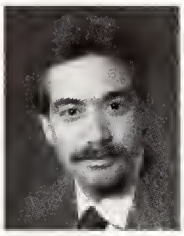
Simeon Aymeloglu is a Distinguished Member of the Technical Staff in the Microprocessor Design Department at Bell Labs. He was responsible for the overall circuit design and coordination of the WE32200's VLSI implementation. His interests include high-performance CMOS design techniques and application-specific integrated circuits. Aymeloglu hold a PhD from the University of Pennsylvania. He is a member of the IEEE.

Thaddeus J. Gabara is a Member of the Technical Staff at Bell Labs and is responsible for timing in the WE32100 and WE32200. He is currently working on high-performance, low-power CMOS I/O transmitter-receiver circuits and methods of decreasing ground bounce in VLSI CMOS chips. Gabara received the BSEE and MSEE from the New Jersey

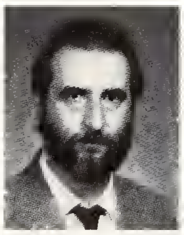
Victor K. L. Huang

James W. Seery

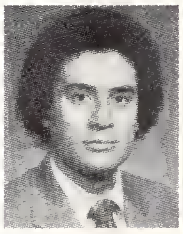
William S. Wu


Saul K. Altabet

Michael J. Killian


Simeon Aymeloglu

Thaddeus J. Gabara


Aaron L. Fisher

Inseok S. Hwang

David W. Thompson

Institute of Technology. He is a member of Eta Kappa Nu, Tau Beta Pi, and the IEEE Electron Devices Society.

Aaron L. Fisher is the supervisor of the Microprocessor Design I group at Bell Labs. His group designs high-performance microprocessors, digital signal processors, and ASICs using full-custom techniques. Fisher received the BSEE from Purdue University and the MSEE from the University of California, Berkeley. He is a member of the IEEE Computer Society.

Inseok S. Hwang is a Member of the Technical Staff in the Microprocessor Design Department at Bell Labs. His major contributions to the design of the WE32200 include the design of the dual-radix ALU. His current interests include advanced processor architecture, fast computer arithmetic, high-speed CMOS design, and pragmatic VLSI testability. Hwang holds the BSEE from Seoul National University and an MSEE and a PhD in electrical engineering from the University of Wisconsin, Madison. He is a member of the ACM and the IEEE.

David W. Thompson is the supervisor of the microprocessor and high-performance CMOS circuit design group. His responsibilities include technical management for CMOS 32-bit microprocessors, as well as microprocessor and peripheral development. He also participates in planning for new integrated circuits and reliability qualification for the microprocessor product line. Thompson received the BSEE from the South Dakota School of Mines and Technology and the MSEE from Stanford University. He is a member of the IEEE, the IEEE Electron Devices Society, and the IEEE Communications Society.

Readers can direct questions about this article to William Wu, AT&T Data Systems Group, Crawfords Corner, Holmdel, NJ 07733.

Reader Interest Survey

Indicate your interest in this article by circling the appropriate number on the Reader Interest Card.

Low 150 **Medium** 151 **High** 152

FEATURE

The Design of the 88000 RISC Family

Advances in silicon technology and RISC design bring cost-effective minimainframe performance with potential upward compatibility to the engineering workstation.

Charles Melear
Motorola, Inc.

The semiconductor industry is currently investigating the value of reduced instruction set computers, or RISCs, for computationally bound applications such as engineering workstations, Unix-based multitasking/multiuser systems, and sophisticated embedded controllers. However, RISCs are not new. Through their use in mainframes, RISCs have long demonstrated performance that is superior to that of superminicomputers and PCs.

In very simple terms, a RISC has a small, streamlined instruction set that operates at higher speeds than other techniques. Dividing a job into small parcels that can be efficiently executed by these simple instructions provides a performance advantage over conventional microprocessor techniques.

Actually, this explanation is a gross oversimplification because the performance of a RISC depends upon its entire support system. For the sake of analogy, consider a simple adder circuit built with emitter-coupled-logic (ECL) technology. The circuit can perform an add operation in just a few nanoseconds—perhaps less (a nanosecond is one billionth of a second). However, problems arise in the process of adding the two numbers to the adder circuit and dealing with the result once it is calculated. If feeding the adder circuit takes a lot longer to perform than the add, you don't gain anything by making a very fast adder circuit. Likewise, if a processor has a very rapid processing speed—but the memory system causes wait states due to access-time restrictions—processor performance becomes a moot point.

Note that the discussion of any RISC system involves much more than an examination of a very high speed computing machine. Designing an economical RISC platform requires integrating the processor into the memory-system architecture as well as generating very efficient, highly optimized machine code.

With these principles in mind, Motorola designed and implemented the RISC 88000 system in high-speed, complementary metal-oxide semiconductor (HCMOS) technology. This technology now allows cost-effective economical fabrication of chips for RISC devices. The 88000 family makes small mainframes economically available as individual engineering workstations.

0272-1732/89/0400-0026$01.00 © 1989 IEEE

Performance

Actual 20-MHz 88000 systems turn in performance ratings equivalent to 14 to 17 VAX million instructions per second (MIPS). They also achieve benchmarks of 38,000 Dhrystones/second. The system's floating-point unit (FPU) is rated at 16.5 million Whetstones/second. Both integer and floating-point instructions can achieve burst rates equal to the clock frequency.

Multiprocessing

Systems with more than one processor offer additional performance without significantly impacting the bandwidth requirements of the memory system. The 88000 architecture specifically includes hardware to facilitate the implementation of multiprocessor systems. Figure 1 is a block diagram of an 88000 system, which functions as an individual processing element.

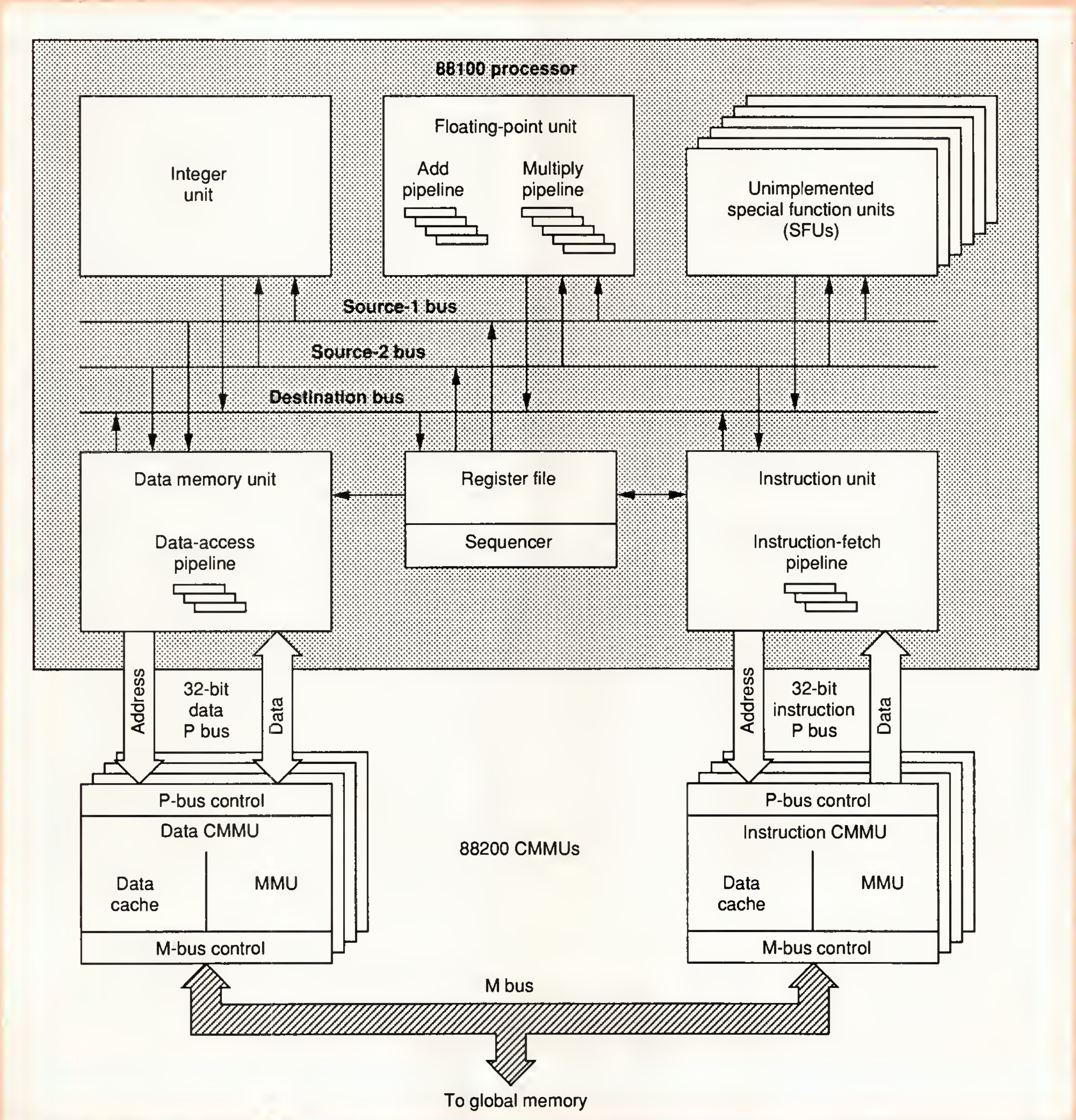


Figure 1. The 88000 system block diagram.

The memory bus (M bus) interfaces two 88200 cache memory management units (CMMUs) to the memory system. The M bus also externally connects to other processing nodes and allows them to share a common memory. Since one CMMU can share a common block of data in a multiple-node system, designers used a hardware method to ensure that all nodes have only the most current data. If one node modifies shared data, all other nodes automatically invalidate the same entry.

Another case arises when data contained in just one cache has been modified. The corresponding external memory location may not have been updated. If another processor tries to access this "stale" data, the node containing the current data automatically updates external memory before allowing the second processor to proceed. Once memory is updated, the original access proceeds correctly. This procedure solves the problem of passing stale data between nodes. Because stale data is automatically invalidated in hardware, the software operating system can devote its attention to controlling the system—rather than checking multiple copies of data for currency.

The CMMUs also provide an arbitration network on the M bus so that multiple-memory bus masters can gain access to global memory.

Object-code compatibility

The designers of the 88000 system established object-code compatibility for future versions of the 88100 processor as a fundamental goal. RISC architectures have a regular nature that tends to make upwardly compatible object code easier to achieve than in architectures that use variable-length instructions. For instance, all RISC instructions, operations, and registers are 32 bits in width. It follows that instruction-set or hardware enhancements will strictly adhere to the internal 32-bit data paths and registers. To allow variable-length instructions with extension words in a RISC architecture would greatly increase the circuit complexity without significantly increasing performance. Maintaining upward object-code compatibility is very important to many users since their most significant investment is in software. When source code is not available for recompilation, object-code compatibility becomes even more important.

Total hardware

As stated, RISC performance depends not only on the speed of the processor but also on its interface with the memory system. Performance also depends on the memory system itself. The total system consists of the 88100 processor and the two CMMUs, which interface with the global memory system. The processor has a Harvard-type architecture, which means that separate processor data/address-bus (P-bus) structures interface to the instruction and data CMMUs. Instructions can only be fetched in the code address space that is addressed by the instruction unit (see Figure 1). No data manipulation can occur in the code address space, primarily because of a read-only instruction-unit data bus. The processor operates on the data contained in the data address space. The data unit cannot fetch instructions from the data memory.

The processor generally requires a new instruction on each clock cycle of 20 MHz or more. This fact places very stringent demands on the memory system. RISC systems do not tolerate memory wait states. One wait state per instruction fetch degrades the machine performance by 50 percent. Therefore, one must either use fast—and very expensive—static RAMs in the memory system or employ some type of caching technique. For the sake of economy, designers chose a caching technique for the 88000 system. This approach allows the use of slower—but more dense—dynamic RAMs for the main memory. A properly sized cache memory can hide most of the wait states from the bulk memory.

Special function units

The 88000 architecture can accommodate more than one functional block that independently executes instructions. The FPU is an example. Special function units (SFUs) sit on the four internal buses. Instructions and operands can be dispatched to and results returned from any one of the execution units as selected by a 3-bit field in the instruction. The field allows logical room for eight SFUs. The integer unit, although it is addressed like an SFU, does not technically fall into that category. But because one of the eight possible codes is taken up by addressing the integer unit, only seven SFUs can be implemented.

Designers chose the FPU as the first and only implemented SFU in the 88000 architecture (Figure 1). All internal registers physically reside in this functional block. Therefore, the FPU could be removed from the 88100 without affecting any other portion of the device. The sole consequence would be the loss of floating-point instructions. This methodology implements a building-block strategy. In the future, system designers can make selections from an SFU library to include in their versions of the 88000 microprocessing unit.

Overview of the 88100

Let's develop an understanding of the internal workings of the processor before further exploring the system aspects of the devices.

The 88100 uses HCMOS logic. It qualifies as a RISC because of its general attributes of single-cycle instruction-execution times and fixed-instruction lengths. It also has a smaller instruction set than conventional

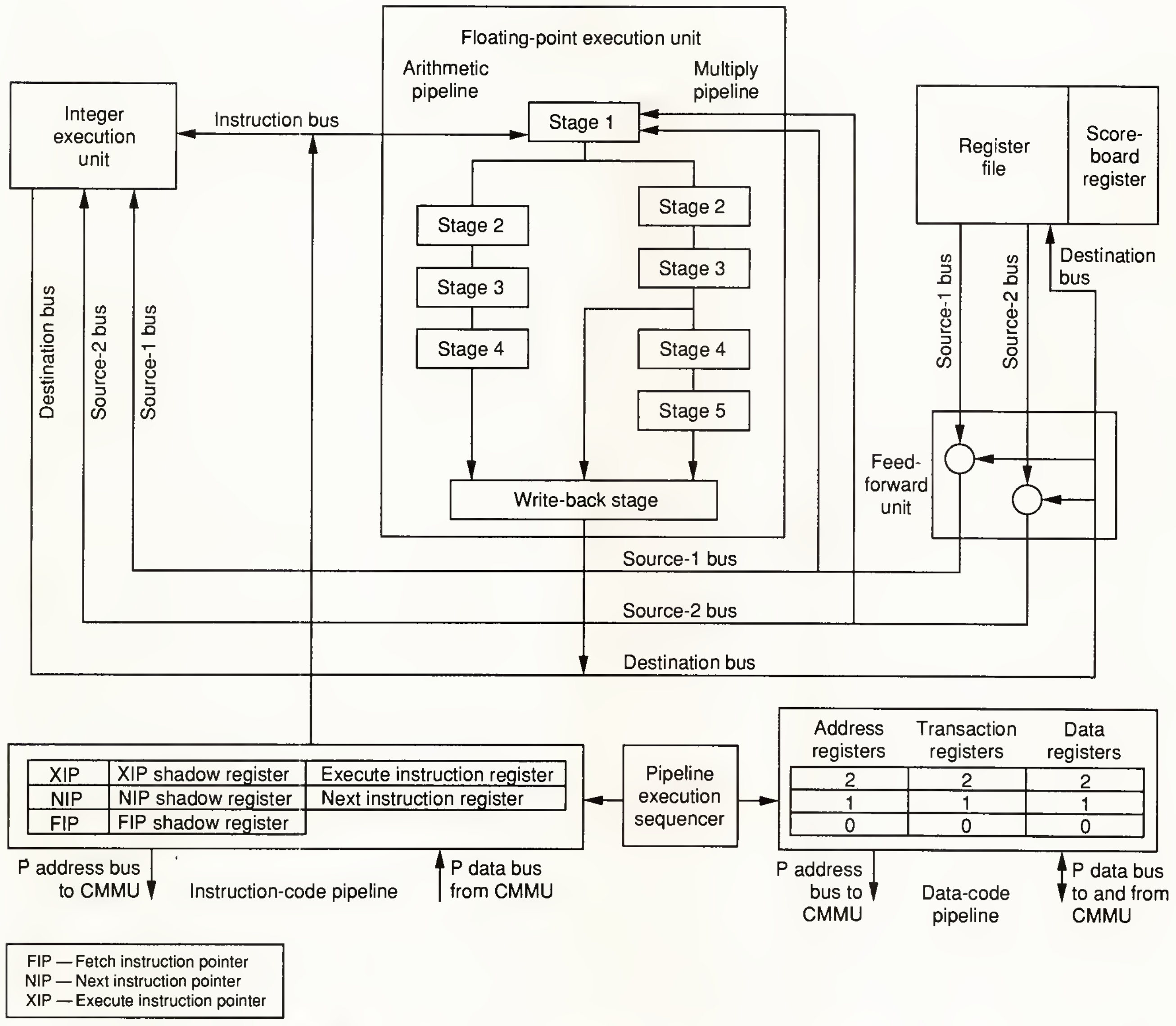


Figure 2. The 88100 processor block diagram.

microprocessor techniques as well as a greatly reduced number of memory addressing modes. In addition to these common RISC attributes, the 88100 implements the just-discussed floating-point arithmetic unit on chip. The FPU is an actual execution unit for the machine that shares its chip real estate.

Figure 2 provides a detailed block diagram of the processor, which employs a dual P-bus system. One P bus serves the instruction memory under instruction-unit control. The second P bus serves the data memory under data-memory unit control. This methodology allows simultaneous instruction fetches along with data-memory transactions.

The machine is parallel in nature, that is, it can work on up to 15 instructions simultaneously. All execution units can perform useful work at the same time. Three memory transactions can progress in the data-memory unit, the instruction unit can fetch one instruction while decoding another, the integer unit can execute an instruction, and the FPU can be executing up to nine instructions—all at once.

Integer-execution unit

The integer unit shown in Figures 1 and 2 executes single-cycle instructions, which essentially include all instructions except Multiply, Divide, Memory-Access, and Floating-Point. Like other execution units, the integer unit connects to four separate buses: instruction,

Table 1.
Exception vectors.

Number	Address	Definition
0	0	Reset (the VBR is cleared before vectoring)
1	VBR+$8	Interrupt
2	VBR+$10	Instruction access exception
3	VBR+$18	Data access exception
4	VBR+$20	Misaligned access
5	VBR+$28	Unimplemented opcode
6	VBR+$30	Privilege violation
7	VBR+$38	Bounds check violation
8	VBR+$40	Integer divide error
9	VBR+$48	Integer overflow
10	VBR+$50	Error
11-113	— —	Reserved for supervisor and future hardware use
114	VBR+$390	SFU1 precise—floating-point precise exception
115	VBR+$398	SFU1 imprecise—floating-point imprecise exception
116	VBR+$3A0	SFU2 precise*
117	VBR+$3A8	Reserved
118	VBR+$3B0	SFU3 precise*
119	VBR+$3B8	Reserved
120	VBR+$3C0	SFU4 precise*
121	VBR+$3C8	Reserved
122	VBR+$3D0	SFU5 precise*
123	VBR+$3D8	Reserved
124	VBR+$3E0	SFU6 precise*
125	VBR+$3E8	Reserved
126	VBR+$3FO	SFU7 precise*
127	VBR+$3F8	Reserved
128-511	— —	Supervisor call exceptions—reserved for user definition

* SFU2 through SFU7 are not implemented. Executing an instruction that is coded for these SFUs causes a precise exception for that SFU.

source-1 and -2 operand, and destination. The source-1 and -2 buses carry operands from the register file or the embedded field of an instruction to an execution unit or SFU. The destination bus returns results to the register file. Instructions dispatch along with the associated operands to the integer unit, and the result returns on the destination bus in one cycle. The system can dispatch a new instruction and receive the result in one cycle. Thus, the integer unit can achieve an execution rate equal to the clock rate.

The overall function of the integer unit is to execute instructions that are dispatched to it by the instruction unit. The integer unit contains dedicated hardware that performs specific functions to complete "difficult" instructions in one clock cycle.

One section calculates numerical results from instructions such as Add and Subtract. Another section is used specifically for bit-field instructions that set, clear, extract, and rotate register fields. A dedicated add unit calculates target addresses for Branch and Jump instructions. As each instruction is fetched, branch-target-calculation circuitry uses part of the instruction operand to calculate a branch target address, whether the just-fetched instruction is a Branch or not. On the next cycle, the sequencer (which controls all instruction and data flow) determines whether or not the instruction is a Branch. If it is, a precalculated target address waits to be used as a fetched instruction pointer (described later). If the instruction is not a Branch, the sequencer simply discards the resultant address calculation.

The instruction pipeline fetches and partially decodes instructions before they are actually dispatched to the appropriate execution unit. During each clock cycle, the pipeline can fetch one instruction, partially decode another one, prefetch any needed operands, and dispatch a third instruction to an execution unit. The pipelined structure is necessary because there is not enough time within one clock cycle to fetch, decode, and execute an instruction. However, dividing the job into sections and using a pipeline technique moves instructions through the instruction pipeline at the rate of one per clock cycle.

The feed-forward unit also speeds program execution. When an instruction dispatches to an execution unit that requires the result of the previous instruction, a problem occurs. There is no time to write the previous result into the register file and make that result available as an operand for the next instruction. When a result is needed, the feed-forward unit solves the problem by taking the result of the previous instruction and routing it on a source-operand bus on the next cycle.

The internal buses of the device carry the instruction, both operands, and the result—all in one cycle. At the speeds that RISCs require, it is not feasible to multiplex internal buses. Therefore, designers implemented separate buses to carry the instruction, operands, and results from the instruction pipeline to the execution units and the register file. The 32-bit bus consumes a great deal of silicon area, but no other acceptable method exists for transferring four 32-bit values in one cycle, as RISC technology requires.

Exception processing

Exceptions can come from a number of sources. Table 1 presents the exception vectors. The exception-vector address can be formed in one of two ways. Concatenating a particular value to the vector base register (VBR) of the integer unit forms hardware exception addresses. For example, say a data-access exception occurs. The fetched-instruction pointer points to the address formed by concatenating the 20-bit VBR with $18, which contains the first instruction of the exception routine. The exception-vector address for instructions such as Trap on Bit Set is formed by taking the 20-bit value in the VBR and concatenating the low-order 9 bits of the Trap instruction followed by three zeros to form a 32-bit address. This address is the location of the first instruction of the exception routine.

Exception processing begins when an external interrupt or any enabled hardware exception occurs. The shadow-freeze (Sfrz) bit of the processor-status register (PSR) sets. Table 2 shows the integer-unit control registers. All trap-time and shadow registers freeze during this cycle, including

- the trap PSR,
- the shadow scoreboard register,
- the shadow registers for the Execute, Next, and Fetch instruction pointers, and
- the shadow registers for the data-memory unit.

All SFUs freeze, that is, instruction processing stops in place. The instruction unit fetches the appropriate instruction in the exception-vector table.

A Trap instruction also initiates exception processing. However, a Trap allows the machine to synchronize itself. That is, before the Trap is actually issued, all memory transactions and floating-point instructions can complete. Then the shadow registers freeze and exception processing continues by fetching the instruction pointed to by the exception vector. (This vector is formed by concatenating the lower 9 bits of the Trap with the VBR.)

Multiple exceptions require some additional processing. Once an exception happens, the Sfrz bit sets and the shadow registers freeze. All SFUs are disabled. If an exception is taken while the Sfrz bit is set, the shadow registers do not reflect the values of the runtime registers. All necessary shadow and general-purpose registers must be stored in external memory to allow nested exceptions. The exception handler software then reenables the SFUs when appropriate, depending upon the cause of the exception. Then software clears the Sfrz bit. This procedure reenables the shadow mode, in which the shadow registers update on a cycle-by-cycle basis and become mirror images of the runtime registers again. This process repeats each time a nested exception occurs.

To return from an exception condition, the shadow registers must contain the appropriate machine context for program return. Setting the Sfrz bit, loading the shadow registers with the desired values, and executing a return from exception (Rte) instruction accomplishes the return. The Rte automatically writes the shadow registers to the runtime registers and clears the Sfrz bit.

Table 2.
Register model of the 88100 integer unit.

Control register no.	Mnemonic	Description of register
0	PID	Processor identification
1	PSR	Processor status
2	TPSR	Trap processor status
3	SSBR	Shadow scoreboard
4	SXIP	Shadow Execute instruction pointer
5	SNIP	Shadow Next instruction pointer
6	SFIP	Shadow Fetch instruction pointer
7	VBR	Vector base
8	DMT2	Transaction 2
9	DMD2	Data 2
10	DMA2	Address 2
11	DMT1	Transaction 1
12	DMD1	Data 1
13	DMA1	Address 1
14	DMT0	Transaction 0
15	DMD0	Data 0
16	DMA0	Address 0
17	SR0	Supervisor storage 0
18	SR1	Supervisor storage 1
19	SR2	Supervisor storage 2
20	SR3	Supervisor storage 3

If further exceptions are prevented by not clearing the Sfrz bit while the current exception processes, it is not necessary to save the shadow registers. They cannot be overwritten. In this case, an Rte automatically returns to the normal context without saving or restoring the shadow registers.

FPU execution

The FPU executes all floating-point instructions as well as Integer Multiplies and Integer Divides. As shown in Figure 3 on the next page, the FPU is a pipelined structure. Thus, the result for the instruction is not ready for several cycles. However, if a result is ready from a prior instruction, it can return to the register file via the destination bus during this cycle. The pipelined nature of the FPU allows a new instruction to

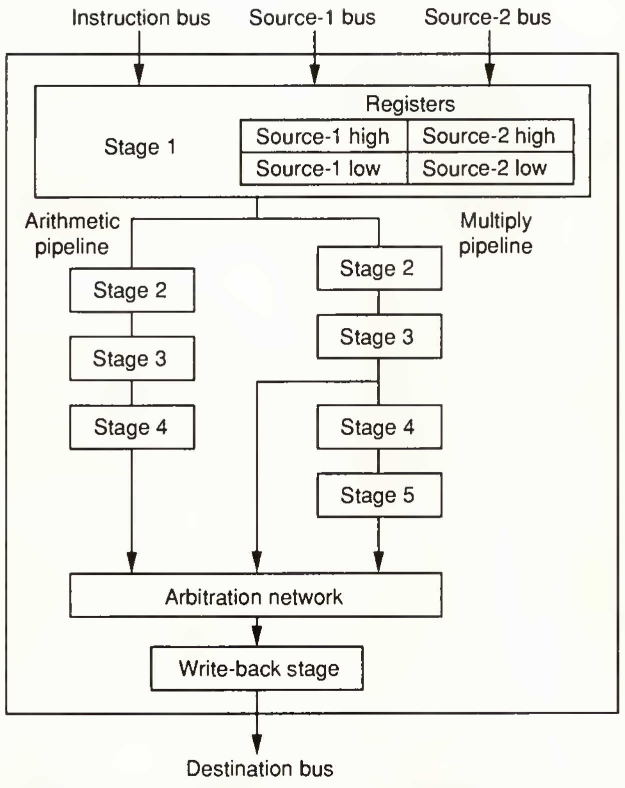


Figure 3. FPU block diagram.

be dispatched on each clock cycle as long as a pipeline stall does not occur. The system can dispatch a double-precision, floating-point instruction on every second clock cycle.

The FPU has two pipelines: arithmetic and multiply. The arithmetic pipeline is used for most floating-point instructions, including Integer Divide and Floating-Point Divide. The multiply pipeline is designed for Integer and Floating-Point Multiplies. Both pipelines begin and end in a common stage. The type of instruction determines which pipeline is used. A single-precision, floating-point instruction cycles to the next stage of the appropriate pipeline on each clock cycle. In the case of double-precision, floating-point instructions, the calculations break into upper and lower words. Each stage in the FPU must first operate on the upper word of the operands and—on the next cycle—on the lower word. For this reason, a new double-precision, floating-point instruction can only start on every second clock cycle.

Write-back arbitration. A result can reach the write-back stage of the FPU in three ways. Integer Multiply instructions flow through stage 3 of the multiply pipeline and then directly to the write-back stage. Floating-point instructions move through stage 5 of the multiply pipeline before reaching the write-back stage. All floating-point arithmetic instructions use all four stages of the floating-point arithmetic pipeline. Three instructions can attempt to deliver results to the write-back stage in the same cycle. In this case, the arbitration network gives priority as follows: Integer Multiply instructions, Floating-Point Multiply instructions, and the floating-point arithmetic pipeline. A long string of consecutive Integer Multiply instructions stalls stage-4 instructions in the arithmetic pipeline and stage-4 and -5 instructions in the multiply pipeline. The arbitration network does not grant access to the write-back stage until the Integer Multiplies complete. Carefully written software generally minimizes the stalling effect of one pipeline on another when data-dependent code sequences occur.

Destination-bus priority. The FPU connects to the destination bus, as do the integer and data-memory units. All three units can have a result ready to drive on the destination bus during the same cycle. In this case, the integer unit has first priority, the FPU second, and the data-memory third. When a long string of integer-unit instructions occurs, the other two units do not get a write slot on the destination bus. Their results have to sit until a write slot occurs.

Various mechanisms exist for assigning write slots to the three units that sit on the destination bus. When an integer-unit instruction issues, it always receives a write slot. If this unit doesn't need the slot—say an integer-unit instruction does not execute during this cycle or the instruction does not generate a result—the FPU can use that slot. The issuing of each floating-point or data-memory-unit instruction creates a write slot if a floating-point result is ready in the write-back stage. Otherwise, this slot goes to the data-memory unit. Whenever the instruction pipeline (discussed later) stalls—causing no instruction to be issued—the system grants a write slot to the highest priority unit that has data waiting to go onto the destination bus.

Floating-point registers. Table 3 illustrates the register model for the FPU. The first nine floating-point control registers (Fcr0-Fcr8) can only be accessed in the supervisor mode and are used for exception processing.

The floating-point user-status and -control registers (Fcr62-63) can be accessed in either user or supervisor mode. The floating-point control registers primarily hold information on instructions that cause exceptions in the FPU. By saving the state of the FPU for the instruction that caused the exception, software can attempt to correct the condition needing service and restart the FPU.

The processor hardware updates the floating-point exception-cause register to indicate the following floating-point exceptions:

- a conversion to integer overflow,
- an unimplemented floating-point instruction,
- a control-register-privilege violation (attempt to access in user mode),
- a floating-point reserved-operand check, and
- a divide by zero.

The following floating-point imprecise exceptions can be signaled as well:

- an underflow,
- an overflow, and
- an inexact condition.

Very simply put, a precise exception is one that signals as soon as the instruction reaches the integer unit or any SFU. For instance, the FPU knows immediately when it receives an unimplemented opcode. In this case,

- the appropriate flag in the exception register sets,
- the exception register points to the offending instruction, and
- the source-1 and source-2 operand high and low registers of the FPU store the operands that were issued with the offending instruction (see Table 3).

User-supplied software routines can handle exception recovery. For instance, an unimplemented opcode can be deliberately inserted in the user's code. The exception handler can decode the opcode portion of the instruction and perhaps run a synthesized instruction (such as trigonometric or hyperbolic) in software.

Imprecise exceptions, on the other hand, do not signal until the instruction has nearly completed. For instance, an underflow (a result with an exponent of -127) does not signal until the result is actually calculated. An underflow is not always fatal. For instance, the number 1101×10^{-127} can also be represented as 11010×10^{-126}. The uncertainty in the last digit may be acceptable for a particular application. It does allow program execution to continue. However, by the time that the offending instruction generates an exception, the instruction pointers no longer point to the instruction. This condition makes it impossible to identify the actual instruction that caused the imprecise exception.

The imprecise operation-type register contains the information that determines what action to take—along with the appropriate instruction information to continue execution. That is, the register contains

- the exponent of the inexact result,
- whether the result is single or double precision,
- the 5-bit opcode that identifies the instruction type,
- which exception handlers became enabled, and
- the destination register for the result.

The floating-point-result high and low registers store the mantissa of the actual inexact result. The floating-point high register also contains information about the rounding modes and the guard, round, and sticky bits that can provide additional bits of accuracy in the result. From this information, software can complete an instruction that caused an imprecise exception.

Table 3.
FPU control registers.

Floating-point control register no.	Mnemonic	Floating-point registers
0	FPECR	Exception cause
1	FPHS1	Source-1 operand high
2	FPLS1	Source-1 operand low
3	FPHS2	Source-2 operand high
4	FPLS2	Source-2 operand low
5	FPPT	Precise-operation type
6	FPRH	Result high
7	FPRL	Result low
8	FPIT	Imprecise-operation type
9-61	—	Unimplemented
62	FPSR	User status
63	FPCR	User control

Register file

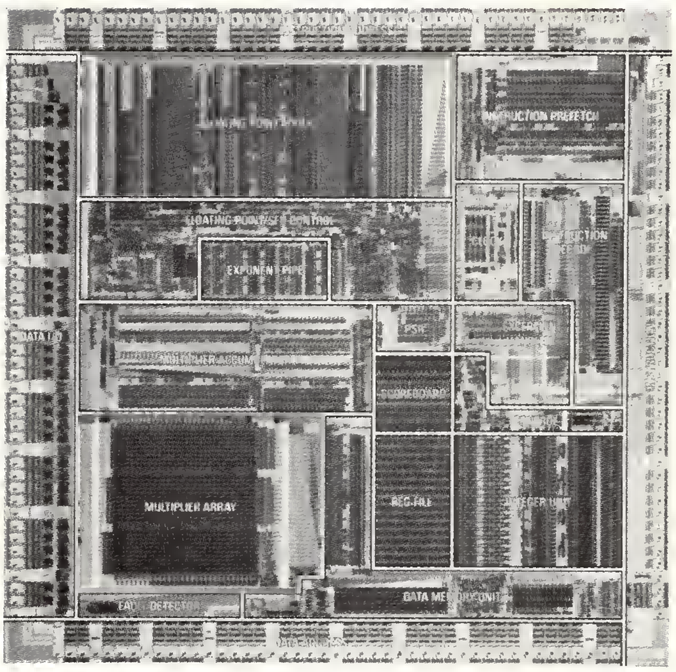
The register file (shown in Figure 2) consists of thirty-two 32-bit, general-purpose registers that source instruction operands and receive the calculated results. The first register, R0, is unique in that the system always reads it as zero and cannot write to it. This condition creates a constant of zero to be used as a source operand, which is convenient for such things as synthesizing single-cycle register-to-register moves. Adding any register to R0 and storing the result in the chosen destination register accomplishes this task.

The R1 general-purpose register also has a special property. The system automatically stores the return address for a Branch or Jump to Subroutine in R1.

The remaining 30 general-purpose registers serve as the source or destination of instruction operands and results. No hardware conventions exist for the use of these registers. Recall that all source operands must come either from embedded fields in the instructions or the register file. Data in external memory must first be loaded into a register in the register file before an instruction can use the data as an operand.

Scoreboard register

This register ensures that a source operand is not fetched from a register that is currently waiting for a result. If operands are available, this hardware scheme



Photomicrograph of the 88100 microprocessor.

lets instructions be dispatched to idle execution units while other instructions are in progress.

The scoreboard register contains 32 bits; one bit corresponds to each register in the register file. When a multiple-cycle instruction is issued, a bit sets in the scoreboard register that corresponds to the register that receives the results of the instruction. The bit clears when the result of the instruction writes to the destination register. Once the scoreboard bit is set, a subsequent instruction cannot use that register for a source operand until the bit clears, which indicates the result of a previous instruction has been delivered. If an instruction reaches the execution stage of the instruction pipe—and tries to fetch an operand from a register with a set scoreboard bit—the instruction pipeline stalls. The instruction does not move to the appropriate execution unit until the scoreboard bit clears. All integer-unit instructions execute in a single cycle. Therefore, integer-unit instructions cannot stall the instruction pipeline.

While instruction-pipeline stalls are inevitable to some degree in any code, properly written software takes advantage of the machine's architecture and arranges instructions in the best possible order to maximize the throughput rate.

Alternatively, the compiler or software writer can deliberately install NO-OPs after multicycle instructions to ensure their completion before another instruction is fetched.

Scoreboarding provides absolute protection from problems that could otherwise arise from out-of-order execution models.

Instruction unit

The instruction unit shown in Figure 1 fetches and partially decodes instructions. All instruction-unit registers are accessible through the integer-execution unit. The instruction unit is a three-stage, pipelined structure consisting of Fetch, Next, and Execute stages.

The Fetch stage consists of the fetched instruction pointer (FIP) and its shadow register (see Figure 2). At the beginning of each cycle, assuming no pipeline stalls or memory wait states occur, the FIP issues a new address to the memory system. This address is either the previous address plus 4 bytes or the target address of the currently executing flow-control instruction.

The second, or Next, stage of the instruction pipeline consists of its instruction pointer (NIP), the NIP shadow register, and the Next instruction register. The FIP of the previous cycle shifts to the NIP, and the corresponding instruction from the memory system returns to the Next instruction register. At this time, the instruction is partially decoded, and any needed operands from the register file are prefetched and prepared for transfer to the appropriate execution unit.

The third, or Execute, stage consists of the its instruction pointer (XIP), the XIP shadow register, and the Execute instruction register. During this stage, the instruction dispatches to the appropriate execution unit. If an exception occurs during any cycle, the shadow registers that maintain real-time copies of their corresponding runtime registers freeze, maintaining the value at the time of the exception as well. This process saves the state of the machine and allows exception processing to begin immediately.

Branch-execution enhancement

The instruction unit handles a special problem associated with flow-control instructions. The sequence for a flow-control instruction is the same as any other instruction through the instruction pipeline. The Branch or Jump is fetched in cycle 1. During cycle 2, the Branch or Jump shifts to the Next-instruction slot and the branch target address is calculated. A new instruction is also fetched during this cycle. During cycle 3, the Branch or Jump goes to the Execute slot and the calculated target address writes into the FIP pointer, which outputs the address onto the external code-address bus. A problem arises: The instruction immediately following the Jump has already been fetched and partially decoded. This instruction normally would not be needed because the program has just been directed to another spot. Therefore, the instruction in the Next slot would normally be invalidated, causing a "hole" in the instruction pipeline.

By using clever programming techniques, one can usually place a useful instruction immediately after a flow-control instruction and obtain a useful result. An "execute next" option provides this flexibility. This option can be enabled or disabled for each individual flow-control instruction (Branch or Jump). It causes the instruction immediately following the flow-control instruction to execute whether the Branch is taken or not.

Now consider the exception-vector table (Table 1). The vectors are aligned on double-word boundaries. Thus the table can hold two instructions per vector. When exception processing vectors to a particular location in the exception-vector table, a flow-control instruction is normally encountered that directs program execution. Placing the first instruction of the exception routine immediately after the flow-control instruction in the table and using the execute-next option lets the flow-control instruction point to the second instruction. Under these conditions, no hole occurs in the instruction pipeline.

Data-memory unit

This unit performs all data-memory transactions (see Figure 1).

When a Load, Store, or Exchange instruction is issued, the sequencer sends the instruction to the data-memory unit. Three register-indirect addressing modes address external memory:

- 16-bit immediate offset,
- indexed offset, and
- scaled-indexed offset.

A dedicated add circuit in the data-memory unit calculates the logical effective address for these addressing modes. This circuit can add a register to a 16-bit immediate value embedded in the instruction, add two registers together, or add two registers together after a scaling operation. It performs the last function after shifting the second register 0, 1, 2, or 3 places to the left to form the effective address. Loads and Stores result in multiple-cycle operations, but they dispatch at the rate of one memory-access instruction per clock cycle because of the pipelined nature of the data unit.

The data-memory unit is a three-stage structure. Each stage contains three registers: address, transaction, and data (see Figure 4). The address register contains the effective address of the memory transaction. The transaction register contains information such as the size of the transaction and its destination

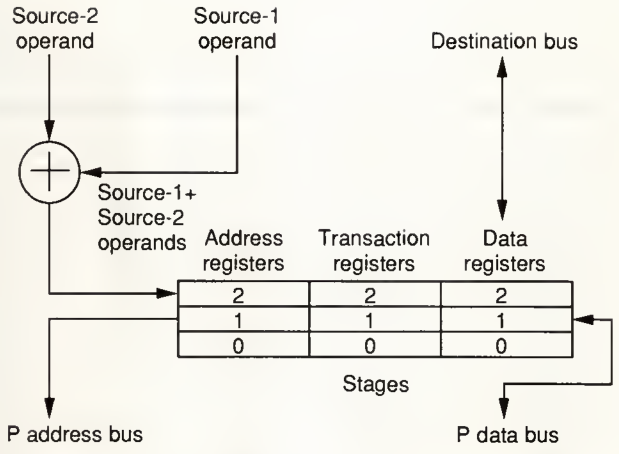


Figure 4. Data-memory unit organization.

register. The data register contains the data to be stored.

In a typical sequence, a store transaction issues the indirect address plus the indexing value to stage 0 of the data pipeline in which the effective address is calculated. The contents of stage-0 registers shift to stage 1 on the next cycle, and the address and data register contents apply to the data-unit address and data buses.

Stage 1 is necessary because there is not enough time to calculate the effective address, apply the address to the external bus, and allow for any appreciable address and data setup times for memory. Stage 1 makes the entire cycle available for a memory access, which greatly reduces the bandwidth requirements of the memory system. During the third cycle, the contents of the stage-1 registers are shifted to stage 2. In the case of a Load instruction, the data returns to the data unit via the data-side data bus.

The only purpose of stage 2 is to maintain a copy of stage-1 information for one additional cycle. This feature allows the implementation of virtual memory systems. If a fault occurs for a memory transaction, the memory system returns the fault signal on the cycle following the access. Thus, when a memory fault or exception signals, the corresponding information about the memory access freezes in stage 2 of the data unit. All memory faults are not lethal. If the exception is caused by a page fault, the handler can find the appropriate section of the program on disk memory, read the required portion of the program into active memory, and modify the memory mapping registers as needed. The execution handler can examine the registers in stage 2 of the data unit and reconstruct the memory transaction that previously faulted. After retrieving the appropriate portion of the program, the memory access can now complete successfully.

Table 4.
Instruction-set summary.

Mnemonic	Description	Mnemonic	Description
Integer arithmetic instructions		Logical instructions (cont'd.)	
add	Add	ext	Extract signed bit field
addu	Add unsigned	extu	Extract unsigned bit field
cmp	Compare	ff0	Find first bit clear
div	Divide	ff1	Find first bit set
divu	Divide unsigned	mak	Make bit field
mul	Multiply	rot	Rotate register
sub	Subtract	set	Set bit field
subu	Subtract unsigned		
		Load/Store/Exchange instructions	
Floating-point arithmetic instructions		ld	Load register from memory
fadd	Add	lda	Load address
fcmp	Compare	ldcr	Load from control register
fdiv	Divide	st	Store register to memory
fldcr	Load from floating-point control register	stcr	Store to control register
fit	Convert integer to floating point	xcr	Exchange control register
fmul	Multiply	xmem	Exchange register with memory
fstcr	Store to floating-point control register		
fsub	Subtract	Flow-control instructions	
fxcr	Exchange floating-point control register	bb0	Branch on bit clear
int	Round floating point to integer	bb1	Branch on bit set
nint	Round floating point to nearest integer	bcnd	Conditional branch
trnc	Truncate floating point to integer	br	Unconditional branch
		bsr	Branch to subroutine
		jmp	Unconditional jump
Logical instructions		jsr	Jump to subroutine
and	AND	rte	Return from exception
mask	Logical mask immediate	tb0	Trap on bit clear
or	OR	tb1	Trap on bit set
xor	Exclusive OR	tbnd	Trap on bounds check
cir	Clear bit field	tcnd	Conditional trap

Instruction set

The RISC instruction set is relatively small in comparison to other kinds of computer architectures. RISC instructions are implemented in hardwired logic. One must add new instructions carefully. Any additions must be absolutely necessary because the logic needed to implement them also greatly impacts circuit density and size. One must evaluate a new instruction in terms of how it can improve overall processor performance. Writing compilers and simulation models of the processor—and evaluating the performance of the instruction set versus the compiler—accomplishes this purpose. Table 4 presents the 88100 instruction set.

RISCs must use many hardware techniques to gain performance, even at the expense of creating larger circuit sizes. Remember that RISC instructions are very elemental; several RISC instructions generally equal one conventional instruction. Therefore, if a RISC is to obtain significant performance improvements, the machine must execute instructions on the highest possible percentage of clock cycles.

As stated, all 88000 instructions are 32 bits wide. The system contains no extension words or instructions shorter than 32 bits. Instruction-set implementation allows for streamlining the internal decoding circuitry. Figure 5 demonstrates the encoding pattern for an Add instruction. Instruction alignment circuitry is unnecessary. Bits 31-26 define which instruction executes.

Bit positions 25-21, 20-16, and 4-0 of an instruction always specify the destination source-1 and source-2 registers. The internal decoding circuitry does not have to locate and align a particular field in the instruction. These fields are always in the same place no matter what the instruction is. This method reduces the amount of circuitry needed to produce a high-performance RISC implementation.

Arithmetic instructions. These instructions include Add, Subtract, Compare, Divide, and Multiply. (Add and Subtract have signed and unsigned forms.) Other architectures can have several variations of particular instructions. For instance, an Add instruction can possess different forms that take advantage of the size of the data field or the location of the data. The RISC methodology does not allow as many forms of instructions because the large circuit size would make an IC unsuitable for manufacture.

Certain hardware techniques create a flexible instruction set. Consider the Add instruction, for which only two forms exist. The first adds the contents of two registers of the register file and delivers the result to a third register. The second adds the contents of a register to a 16-bit immediate field embedded in the instruction. A dedicated bit in the instruction enables/disables the overflow exception. Another bit causes the carry bit to be included/not included in the calculation. Yet another bit causes a carry bit to be generated/not generated. Implementing one instruction allows eight basic variations of Add: signed and unsigned, with or without underflow/overflow, and with or without carry.

The immediate form of Add is always without carry. This exact same scheme generates the eight basic forms of the Subtract instruction.

Condition codes. The Compare instruction calculates these codes. It compares two registers with one another or one register with a 16-bit immediate value embedded in the instruction modes. The results are placed in a destination register. Figure 6 shows the encoding pattern for the resultant condition codes.

Condition codes are not explicitly generated as each instruction executes. Condition codes need to be calculated only when they are used for a following conditional flow-control instruction such as a Branch on Condition. The circuitry needed to generate condition codes in machines with out-of-order execution models is quite complex. Therefore, designers implemented the Compare instruction to explicitly generate condition codes when needed and execute a Branch on Bit Set/Clear to emulate the desired Branch instruction.

Logical and flow-control instructions. Bit-field instructions extend RISC instruction sets designed for prior machines. Special hardware within the integer unit facilitates the execution of bit-field instructions.

111101	D	S1	011100	I	O	000	S2

Add R7, R8, and R1. Add contents of R8 to R1 and place results in R7.

111101	00111	01000	011100	1	1	000	00001

D	5-bit field specifying destination register
I	Enable/disable carry in
O	Enable/disable carry out
S1	5-bit field specifying source-1 register
S2	5-bit field specifying source-2 register

Figure 5. Add-instruction encoding.

111101	D	S1	011111	00000	S2

31	11	10	9	8	7	6	5	4	3	2	0
	hs	lo	ls	hi	ge	lt	le	gt	ne	eq	

Compare result written to destination register.

D	5-bit field specifying destination register
eq	Equal
ge	Greater than or equal
gt	Greater than
hi	Higher than
hs	Higher than or the same
lt	Less than
le	Lower than or equal
lo	Lower than
ls	Lower than or the same
ne	Not equal
S1	5-bit field specifying source-1 register
S2	5-bit field specifying source-2 register

Figure 6. Compare-instruction encoding.

The fields on which these instructions operate can be of any width and located anywhere in the word. Bit-field hardware can clear, set, extract, and insert fields into registers. This hardware can essentially perform a single-cycle shift of any number of bits to a field of any width. The only limitation is that the amount of the shift plus the width of the affected field must be less than the width of the 32-bit register. Also, the Rotate instruction always rotates the entire contents of a 32-bit register, that is, the field width is always 32 bits.

Floating-point instructions. These instructions are also a modern extension of typical HCMOS RISC architectures. RISCs generally implement only the most basic arithmetic, logical, and flow-control instructions. Floating-point instructions vary from typical RISC techniques because of their multiple-cycle execution times. In this particular case, designers implemented enough hardware to perform floating-point arithmetic in a pipelined, sequential fashion. Placing the FPU on the internal silicon buses (which can provide a new floating-point instruction on every clock cycle) yields superior performance. In fact, the variance from standard single-cycle execution is well worth the additional cost that results from additional circuit size.

Although this article has described a data-processing machine that inhabits the very upward limit of HCMOS-microprocessor design and silicon-processing technology, remember that RISCs gain performance because developers fine-tune the entire system. The system must contain enough of the right instructions to allow compilers to generate efficient code. It means nothing if the speed of the processor doubles but requires four times as many instructions. A memory system that cannot supply instructions to the processor without wait states gains nil. Designers must interface each subsystem of the chip design as efficiently as possible with all other subsystems.

RISC systems have actually been around for some 25 years, but their primary application has been in mainframes built with ECL technology. Present silicon-wafer processing techniques allow very large systems to be built in HCMOS technology.

If history is any guide, some of the emerging RISCs will yield additional processing power at an increasingly cost-effective level to bring mainframe performance to the desktop market. Perhaps the real challenge is how to put that power to work in new and exciting applications.

Readers can direct questions concerning this article to the author at Motorola, Inc., 6501 William Cannon Drive West, MS/OE33, Austin, TX 78735-8598.

Reader Interest Survey

Indicate your interest in this article by circling the appropriate number on the Reader Interest Card.

Low 153 **Medium** 154 **High** 155

Dynamically Changing the Logical Behavior of a Microcomputer Interface

Don't use separate encoded programs to achieve multiple output combinational circuits. Use this Boolean data processor, and you can produce synchronous sequential circuits also.

Ioan Dancea
Quebec University at Hull

In programmable controllers, as in other practical applications, software partially implements the logical behavior of an interface. Software especially proves useful when this behavior must be changed frequently and does not require arithmetic and/or parallel processing. Though such a hardware logic implementation is always much faster than a sequentially stored program, the stored program is more flexible, because its modification is easily accommodated.

Direct logic simulation method

Industrial programmable controllers and hardware simulators currently use the software method of direct logic simulation. In this method programs contain the logical functions of several Boolean variables that are encoded directly. Each combinational circuit expressed by a group of Boolean equations requires an independent program. There are no restrictions as to the form of these equations (sum-of-products, product-of-sums, or a combination of them, possibly with brackets).

Traditionally, programmers write in assembler language a program to solve equations of this type and use standard wordwide logical instructions and conditional jumps. The systems integration is simple, but it requires considerable programming effort and inefficient memory utilization. Even for a moderately complex practical circuit, the resulting code is cumbersome and error prone. Furthermore, execution times vary widely with input data.

To improve the program's efficiency in the direct logic simulation method, Intel designed several microcontrollers (for example, 8051, 8096) capable of direct bit-test operation. The controllers provide an instruction that implements this operation[1] and replaces the move/mask/conditional jump sequence. The main disadvantage of the direct logic simulation method is the need to include an independent program for any simulated combinational circuit.

Decision diagram method

In the last few years a great deal of work has been achieved to develop the method of decision diagrams and equivalent techniques.[2-4] As a result, we can construct a binary decision diagram from the information contained in a truth table. The diagram implements a multiple output combinational circuit using an

Applying Decision Diagrams to Simulate a Combinational Circuit

As an example of an implementation of the decision diagram method, consider the multiple output combinational circuit characterized by:

• the input variables *a*, *b*, *c*, and *d* and the outputs *X*, *Y*, and *Z*, each group packaged into one byte (Figure A); and
• to describe the circuit, the Boolean equations

$$X = a.b + c.d,\quad Y = a.b + \bar{c}.d, \text{ and } \quad Z = a.b + \bar{c}.\bar{d}. \qquad \textbf{(A)}$$

The system in Equation A allows the development of the decision diagram shown in Figure B. The program that corresponds to this diagram contains, for each node (test block), a mask instruction followed by a conditional jump instruction.

If we suppose the values of input variables to be $a = 1, b = 0, c = 0$, and $d = 1$, the input byte would be in = 09*h*. The darker line indicates the nodes passed during the execution of the program to give the output byte out = 02*h*.

Keeping the same physical configuration for input/output communication, any change, even a minor one in the Boolean equations shown in Equation A, imposes a change of the decision diagram and consequently of the program. For instance, if we replace Equation A by

$$X = a.b + c.d,\quad Y = a.b + c.\bar{d},\quad Z = a.b + \bar{c}.d, \qquad \textbf{(B)}$$

the corresponding decision diagram would be that shown in Figure C. In this figure, the darker line through the nodes indicates the execution of the program for the same values of input variables, to obtain the output out = 01*h*.

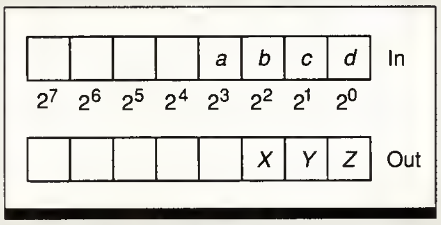


Figure A. The representation of input and output bytes.

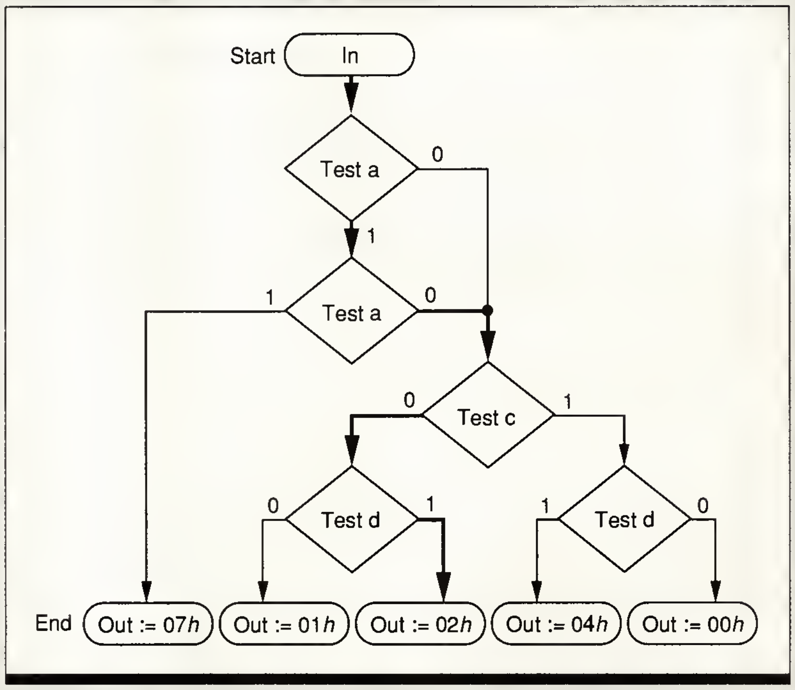


Figure B. The decision diagram of Boolean equations in Equation A.

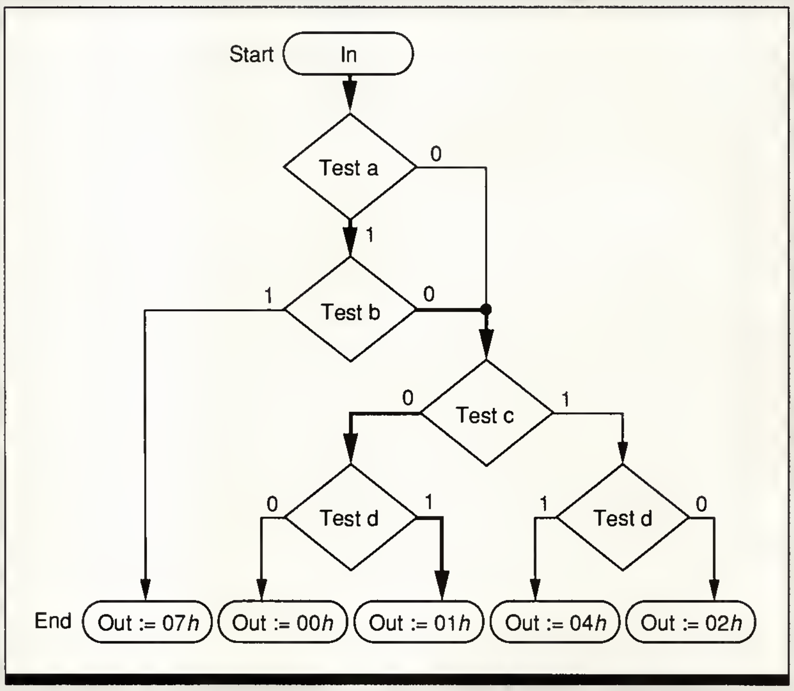


Figure C. The decision diagram of Boolean equations in Equation B.

algorithm that executes a sequence of decision tests over the values of input variables. When considering a word-wide logical instruction, the diagram provides at least one node for each individual input Boolean variable consisting of a mask instruction followed by a conditional jump instruction. Consequently, the program implementation of a decision diagram is a sequence that simulates the interconnection decision nodes. For such a program, it can be shown that:[2]

- the total number of nodes is at the most equal to $2^n - 1$, where n represents the number of input variables; and
- for the software simulation of a multiple output combinational circuit, the program must execute at the most n nodes.

The important advantages of programs using decision diagrams are their structured organization and generally faster execution than those using the direct logic simulation method. Recently, Telfer et al. reported a performant binary decision machine.[5] But, like the direct simulation method, in BDM each multiple output combinational circuit requires a separate encoded program. For further information, see the accompanying box.

A product terms method

In our work[6] we proposed, in the first stage, an alternative solution for the software implementation of the Boolean equations named the *product terms method.* This method features the use of a single program to implement any multiple output combinational circuit. To make the distinction between different circuits, a block of data defines each circuit.

In the second stage, we extended the product terms method to synchronous sequential circuits in which we use two tables. One table determines the next state of the combinational circuit, and the second determines the outputs.

The manual development of the tables, which characterize a group of Boolean equations, is an operation demanding great care, one that is error prone. Consequently, in the third stage,[7] we proposed an expert system for developing these tables from the sum-of-products form of Boolean equations. The expert system is well suited to the product terms method since it is much easier to produce automatically a block of data than a program. In addition to the development of the tables, the expert system can consult a database and identify the circuit that is described by Boolean equations when such equations are already stored in memory. Consequently, we developed a method for changing dynamically the logical behavior of an interface in which different behaviors can be described by different Boolean equations.

Finally, to prove and test the performance of the product terms method and of the expert system, we developed a real interface with its hardware driver on an IBM PC.

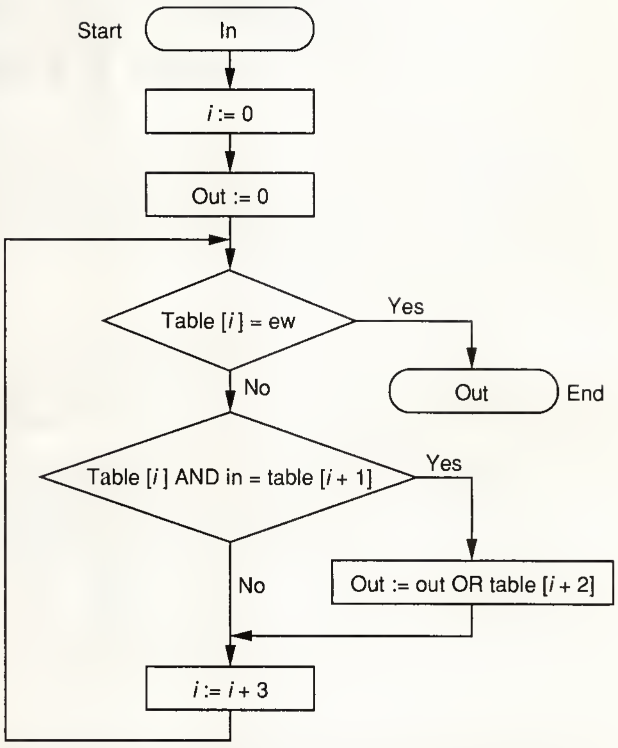


Figure 1. The flowchart of the product terms method.

Multiple output combinational circuits

To apply the product terms method for software implementation of a multiple output combinational circuit, we begin with the sum-of-products form of the Boolean equations.[6] Considering these equations, we note that a prestored table expresses the circuit behavior. To construct this table, we expand each product term to form three memory words. The first, called the *mask word*, corresponds to the association of the variables that compose a product term. Next is the *product word*, which represents the association of the variables that require a logical 1 as output. The third, called the *output word*, indicates the contribution of the product term to the outputs.

During the development of the algorithm, the word that packs the values of the input variables is sequentially compared with the words from the table to produce the word that expresses the outputs. To terminate the execution of the algorithm, we introduce a special *end word.* With these specifications, the algorithm itself is described in the flowchart of Figure 1. Here,

- table [i] contains $3n + 1$ words for n product terms,
- *in* represents the input word,
- *out* expresses the outputs (at the end), and
- *ew* (end word) is a special word stored at the end of the table.

Applying the Product Terms Method to Simulate a Combinational Circuit

As an implementation example of the product terms method, consider the same multiple output combinational circuit discussed in the decision diagram method. Calculate weight indexes for the input Boolean variables, respectively for the outputs:

$a \leftarrow 8$; $b \leftarrow 4$; $c \leftarrow 2$, and $d \leftarrow 1$;
$X \leftarrow 4$; $Y \leftarrow 2$; and $Z \leftarrow 1$.

Using these weight indexes, we associate three words to each product term (Table A) of Boolean equations (Equation A). Consequently, the table seen in Figure D expresses the circuit behavior.

Now, to simulate the circuit, we use the program developed from the algorithm presented in the flowchart of Figure 1 and the table that indicates the circuit behavior.

Any change in the Boolean equations requires only a change of table, but not a change of program. If we consider the Boolean equations in Equation B and keep the same physical configuration for input/output communication, the table that expresses the circuit behavior becomes that shown in Figure E.

Thus, the main characteristic of our method is the use of one and only one program to simulate any multiple output combinational circuit in which each circuit is characterized by its own table. This versatility comes at the expense of an increased execution time. Precisely, for every product term, we go through one loop in the flowchart of Figure 1, which expresses the algorithm. For instance, considering the circuit described by Boolean equations (Equation A) and the input byte in = 09*h*, we have on

- the first pass, out = 00*h*,
- the second pass, out = 00*h*,
- the third pass, out = 02*h*, and
- the fourth pass, out = 02*h*.

Even if we obtained the correct output in the third pass, the execution of the algorithm would continue until the end word was encountered in the table.

Table A.
The words associated with product terms.

Product term	Mask word	Product word	Function word
$a.b$	12	12	7
$c.d$	3	3	4
$\bar{c}.d$	3	1	2
$\bar{c}.\bar{d}$	3	0	1

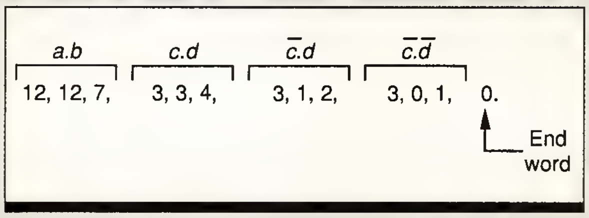


Figure D. The product terms table of the Boolean equations in Equation A.

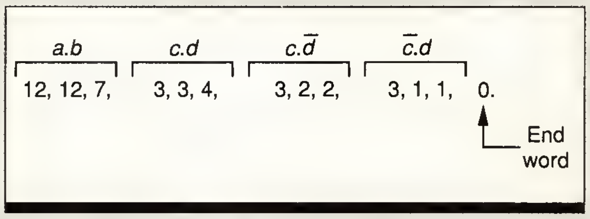


Figure E. The product terms table of the Boolean equations in Equation B.

Obviously, to minimize the CPU time and the required memory, the table describing the behavior of any combinational circuit must contain the minimum number of product terms. The goal in minimization is to find the smallest number of product terms (list of cubes) that can cover all the outputs. We chose the McBoole[8] minimizer, because it is faster for functions having a moderate number of prime cubes and up to 20 input variables. The minimization process is performed outside of the method. The final Boolean equations considered must always be in the minimized form. Further explanation appears in the accompanying box.

Synchronous sequential circuits

One can always transform a general (Mealy) sequential circuit into a Moore sequential circuit in which the outputs depend on the present state only. For our purpose, we represent any sequential circuit as a Moore machine (Figure 2). Here, combinational circuit 1 determines the outputs, and combinational circuit 2 determines the next state.

The nature of the logical flip-flops used in the memory part influences the structure of combinational circuit 2. The simplest solution uses D synchronous flip-flops.

Then, for a sequential circuit, one must use the proposed algorithm twice: first for combinational circuit 2, which determines the next state, and second for combinational circuit 1, which determines the outputs. We always apply the algorithm in this order, since the next state represents the input variables to the combinational circuit that defines the outputs. We store the present state at the beginning of the table that characterizes combinational circuit 2, using a special word placed before the first group of three words that correspond to a logical product. When the table is loaded, we fill that word with the value of the initial state. To generalize the data format, we added a word in the first place for each table representing a pure combinational circuit, but, for this kind of circuit, the word does not have any significance. In this way, for n product terms, a table that characterizes any group of Boolean equations has $3n + 2$ words.

See the box on the next page for further details.

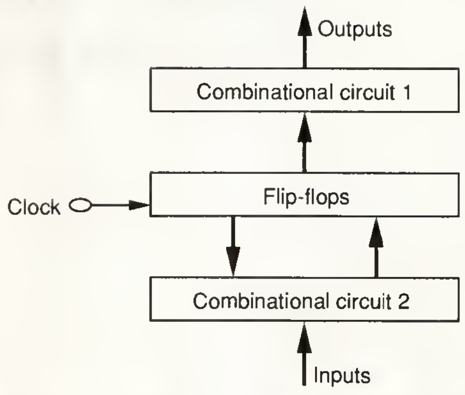


Figure 2. The Moore sequential circuit.

The expert system

As mentioned in the introduction, the manual construction of the table describing any combinational circuit is cumbersome and error prone. Consequently, we developed an expert system[7] to generate the table used in the product terms method by interpreting the symbolic Boolean equations supplied by the user. Really, the proposed expert system, shown in the general flowchart of Figure 3, can also implement other tasks. The program, written in Turbo Prolog, runs on an IBM PC.

The first part interacts with the user and offers a choice to either

• introduce from the keyboard the behavior of the simulated circuit in the form of Boolean equations of the sum-of-products or

• consult a database that has prestored such equations for a number of physical structures.

In the first case, the user supplies the equations. For the other choice, the user consults a *knowledge base* using a classical *backward-chaining inference engine*.[9] The inference engine represents the second part of the expert system. It works in a query mode, matching the user's answer to determine the nature of the structure. The knowledge base contains the parametric description of each logical structure having a file in the database.

Once the logical structure is identified, the system selects from the database the appropriate file describing its behavior, that is, the number of combinational circuits, the input variables, the outputs, and the equations. Usually, such a file contains the description of several combinational circuits intended to characterize the simulated part of the interface. This file selection is the third part of the expert system. We mention once again that the knowledge base is a file in which each structure is described by parameters, and the database contains a separate file for

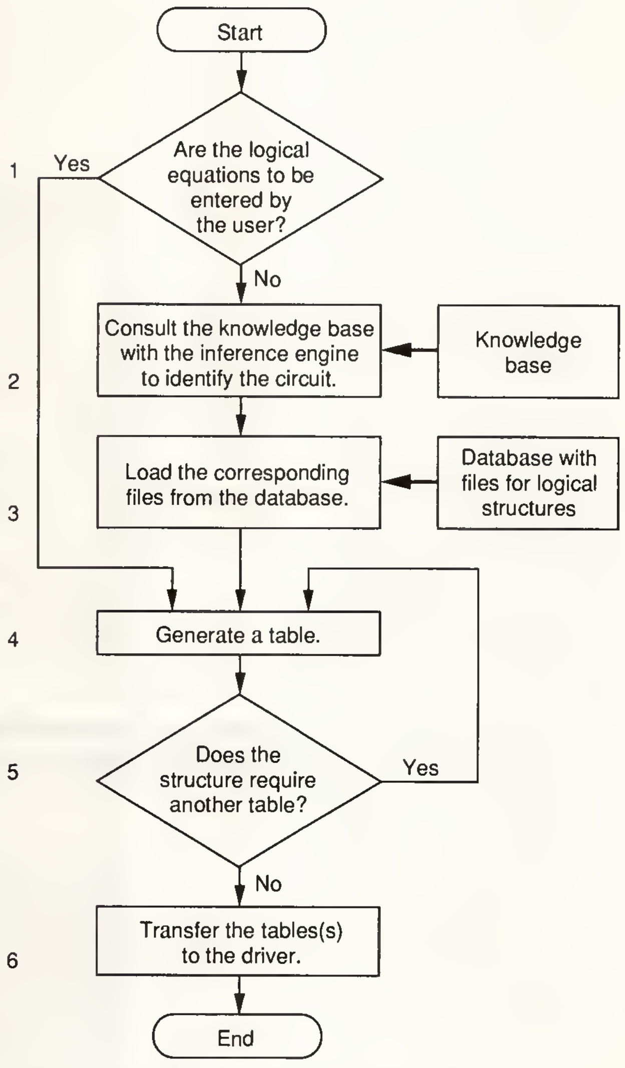

Start
1
Yes
Are the logical equations to be entered by the user?
No
2
Consult the knowledge base with the inference engine to identify the circuit.
Knowledge base
3
Load the corresponding files from the database.
Database with files for logical structures
4
Generate a table.
5
Does the structure require another table?
Yes
No
6
Transfer the tables(s) to the driver.
End

Figure 3. The general flowchart of the expert system.

Applying the Product Terms Method to Simulate a Sequential Circuit

Figure F represents a hardware solution of an electronic dice-game sequential circuit. The circuit simulates two dice; the states of a die appear in Table B. The states correspond to the successive images of the die expressed by six LEDs.[10]

A counting process can successively change the state of the dice. To throw the dice, the user pushes a button which stops a high-frequency modulo 36 counter. In other words, the user generates a random number between 0 and 35, because the dice-game circuit may only have 36 possible states. From this random number, the counting process determines the next state of the dice. For instance, if the dice are in state 5:3 and the generated number is 13, the next stable state will be 1:4. This state is determined by the counting process: 5:3, 5:4, 5:6, 6:1, 6:2, 6:3, 6:4, 6:5, 6:6, 1:1, 1:2, 1:3, and 1:4. It can be seen that, in the counting process, the first die changes only when the second die goes from value 6 to value 1. With these functional descriptions, we determined the equations of the combinational circuit 2, which establishes the next state:[6]

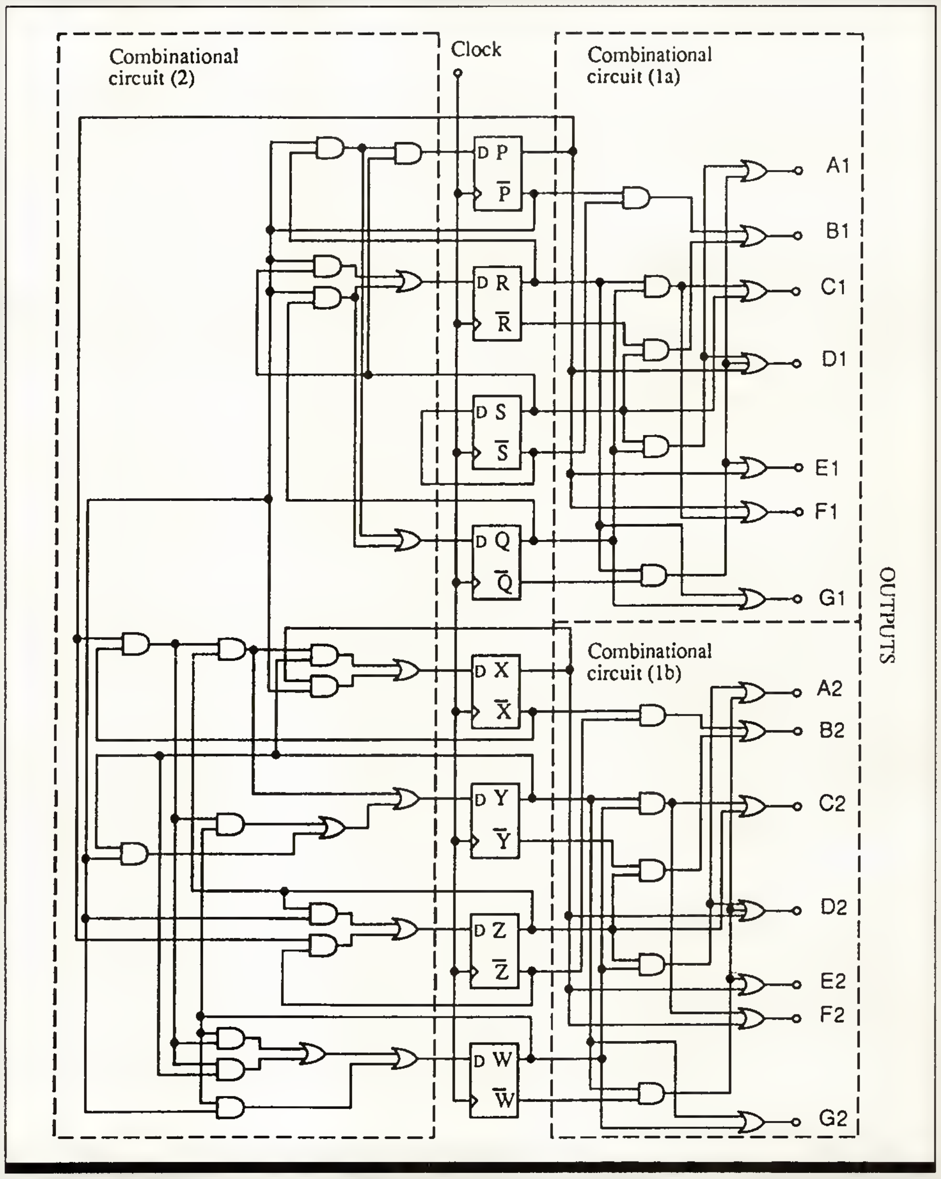


Figure F. An electronic dice game.

$$\begin{aligned}
W &= \bar{p}w + p\bar{x}y + p\bar{x}w;\\
Z &= \bar{p}z + p\bar{z};\\
Y &= \bar{p}y + p\bar{x}z + p\bar{x}w;\\
X &= \bar{p}x + p\bar{x}yx;\\
Q &= \bar{p}r + \bar{p}q;\\
S &= \bar{s};\\
R &= \bar{p}s + \bar{p}q;\\
P &= \bar{p}rs.
\end{aligned} \qquad \text{(C)}$$

In these equations w, z, y, x, q, s, r, and p characterize the present state, and W, Z, Y, X, Q, S, R, and P, the next state. When we develop a table from Equation C for the method of product terms, we must introduce a special word in the first position to indicate the initial state. If, for our example, one considers the initial state 6:6, this value represents in *code die* 0BB hex or 187 decimal.

Two seven-segment indicators display the stable state achieved by the counting process. Consequently, between the flip-flops and these indica-

Table B.
The code die.

State	Q	S	R	P
1	0	1	0	0
2	0	0	1	0
3	1	1	0	0
4	1	0	1	0
5	1	1	1	0
6	1	0	1	1

tors, we introduced two identical decoders, type code die, to seven segments. For such a decoder, we took into consideration all the redundant combinations. With these specifications, we determined the equations of the decoder, the combinational circuit 1a:[6]

$$\begin{aligned} &A = sq + r\overline{q};\\ &B = \overline{r}\overline{s} + \overline{p}\overline{s};\\ &C = rq + s;\\ &D = sq + r\overline{q} + p; \qquad \textbf{(D)}\\ &E = r\overline{q} + p;\\ &F = rq + p;\\ &G = q + r. \end{aligned}$$

The combinational circuit 1b has the same logical structure as circuit 1a and, consequently, the same table in the product terms method. The input variables $q, s, r,$ and p are replaced by $w, z, y,$ and x. For this example, see the generated tables in Figure 9.

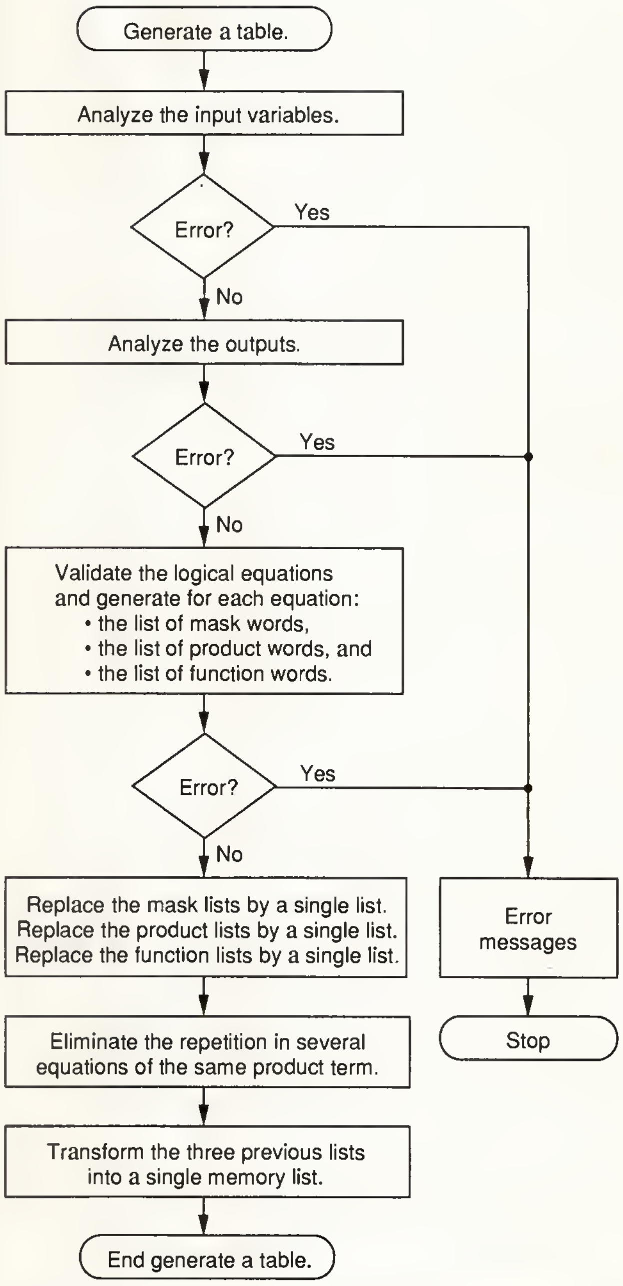

Generate a table.
Analyze the input variables.
Error?
Yes
No
Analyze the outputs.
Error?
Yes
No
Validate the logical equations and generate for each equation:
• the list of mask words,
• the list of product words, and
• the list of function words.
Error?
Yes
No
Replace the mask lists by a single list. Replace the product lists by a single list. Replace the function lists by a single list.
Error messages
Stop
Eliminate the repetition in several equations of the same product term.
Transform the three previous lists into a single memory list.
End generate a table.

Figure 4. The flowchart of the block, Generate a table.

each circuit having its behaviors expressed by Boolean equations.

The most important and intricate part of the expert system is the fourth one. For any multiple output combinational circuit, this part generates, from the Boolean equations, the table used by the product terms method. Here, we fully exploit the power of the Prolog language to process lists. See the flowchart of Figure 4 for details. Initially, the expert system validates the input variables, the outputs, and the symbolic representation of the Boolean equations, checking that the following conditions are satisfied:

- input variables appear as lowercase letters;
- the number of input variables does not exceed word length (eight bits in the actual implementation);
- the same letter is not repeated in the input string;
- outputs appear in uppercase letters;
- the number of outputs does not exceed word length (eight bits in the actual implementation);

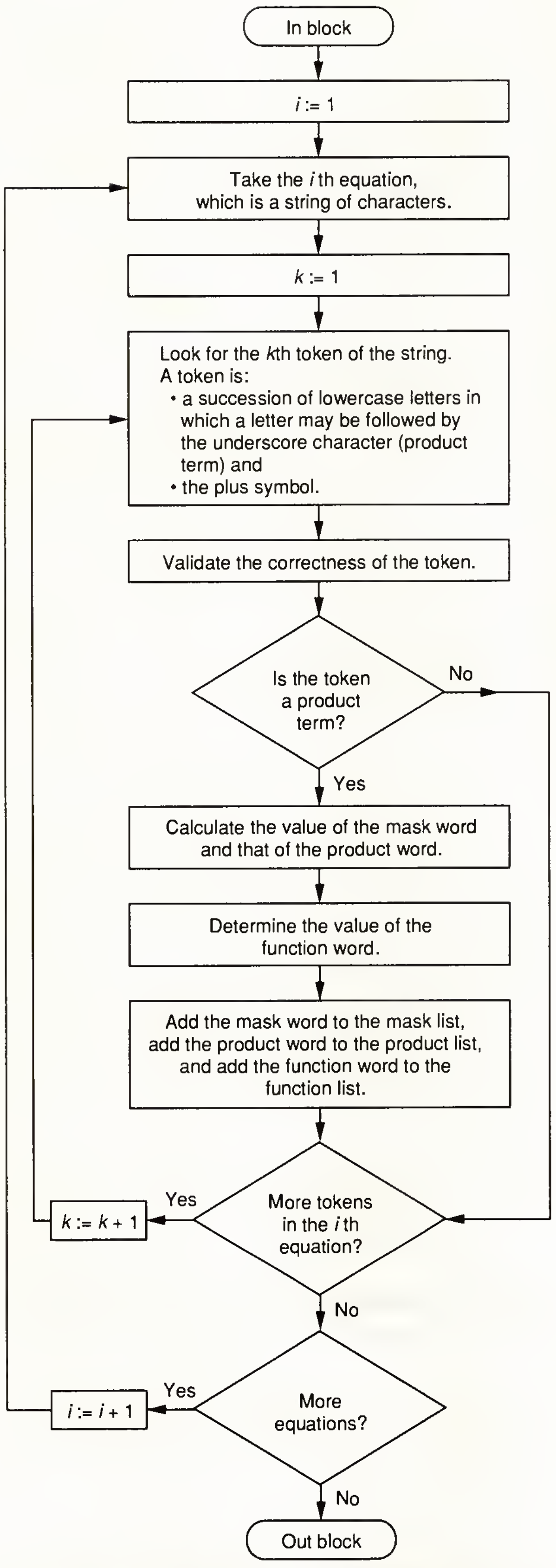

In block
i := 1
Take the *i*th equation, which is a string of characters.
k := 1
Look for the *k*th token of the string. A token is:
• a succession of lowercase letters in which a letter may be followed by the underscore character (product term) and
• the plus symbol.
Validate the correctness of the token.
Is the token a product term?
No
Yes
Calculate the value of the mask word and that of the product word.
Determine the value of the function word.
Add the mask word to the mask list, add the product word to the product list, and add the function word to the function list.
More tokens in the *i*th equation?
Yes
k := *k* + 1
No
More equations?
Yes
i := *i* + 1
No
Out block

Figure 5. Detailed part of the block, Validate the logical equations.

• the same letter is not repeated in the output string;
• an equation contains only lowercase letters, the underscore character (used to denote negation, that is, not *a*, is represented by *a*_), and the plus symbol;
• the same variable is not repeated in a product term;
• a product term does not begin with the underscore symbol;
• underscore symbols do not follow one another;
• plus symbols do not follow one another;
• an equation does not begin with the plus symbol; and
• an equation does not end with the plus symbol.

If all these conditions are achieved, the system generates the lists of mask words, the lists of product words, and the lists of function words. Next, the system transforms a group of lists of the same type into a single list and eliminates the repetition of the same product term. Finally, it combines the three lists (mask, product, and function) into a memory list that expresses the behavior of the combinational circuit. For more clarity, Figure 5 shows in detail the block that generates the three lists for each group of Boolean equations.

The fifth part of the general flowchart imposes the repetition of the block, *Generate a table*, if the simulated structure contains more than one combinational circuit. Of course, this repetition is done for a sequential circuit.

The task of the sixth and last part of the expert system, written in assembly language, is to transfer the memory table(s) to the driver controlling the interface. We mention again that our method can work correctly only if the input/output communication with the physical part of the interface remains unchanged. Once these tables are transferred, the driver works concurrently with other programs to share the processor time.

A real interface and driver

To prove and test the performance of the product terms method and of the expert system, we tied a simple interface, mainly implemented with an Intel 8255A programmable parallel communication circuit, to an IBM PC. A hardware driver, which is a memory-resident routine written in assembly language and accessed through the interrupt mechanism, controls the interface. Figure 6 shows a simplified image of the interface and its driver. Figures 7 and 8 display the flowcharts of the two main parts of the driver.

The installation part, which is discarded after the installation is completed, programs the external 8255A circuit and the two IBM PC internal circuits (the 8253 timer and the 8259 interrupt controller).

The resident part of the driver simulates a structure composed of a memory register and three combinational circuits, each one defined by a group of Boolean equations. By changing the logical expression of any combinational circuit, we change the logical behavior of the structure. Note that the input/output communication must remain unchanged. One can imagine and implement a completely different driver for the same external interface or can

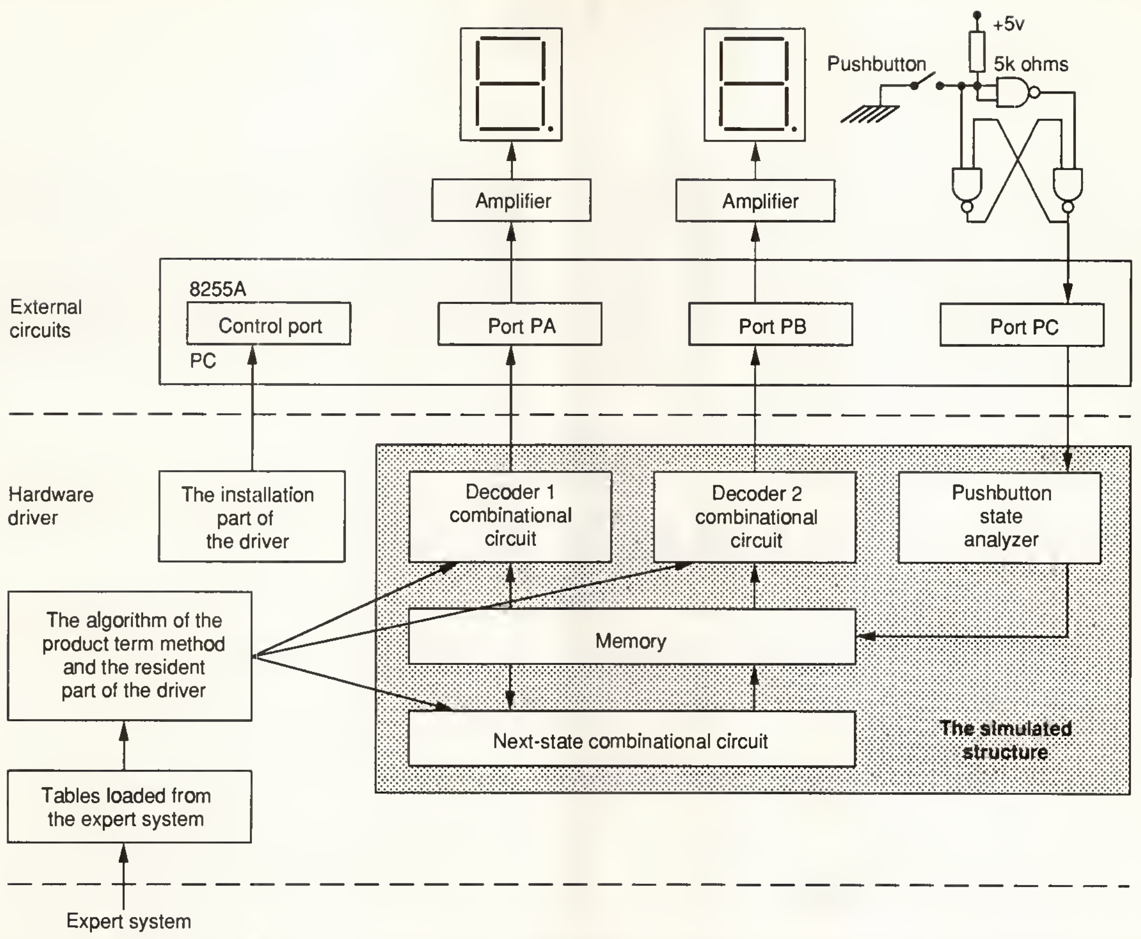

+5v
Pushbutton
5k ohms
Amplifier
Amplifier
External circuits
8255A
Control port
PC
Port PA
Port PB
Port PC
Hardware driver
The installation part of the driver
Decoder 1 combinational circuit
Decoder 2 combinational circuit
Pushbutton state analyzer
The algorithm of the product term method and the resident part of the driver
Memory
Next-state combinational circuit
The simulated structure
Tables loaded from the expert system
Expert system

Figure 6. A simplified view of the interface.

change the external interface and its driver, but the driver can only simulate one logical structure at the time.

To change the logical behavior, we change the information memorized in the tables used by the product terms method. To do this, we use the expert system. We mention that the complexity of the circuit that can be realized is limited by the hardware structure of the physical part of the interface and the logical structure of the driver, but not by the expert system. Consequently, the largest part of the expert system does not change when we modify the external interface and/or the driver.

Coming back to our implementation, the resident part of the driver can choose between several structures:

- an eight-bit direct binary counter with two decoders,
- an eight-bit reverse binary counter with two decoders,
- a two-digit direct BCD counter with two decoders,
- a two-digit reverse BCD counter with two decoders,
- a minute chronometer, and
- a dice game (the equivalent hardware solution appears in the earlier Figure F).

All three combinational circuits of the simulated structure are not always required. Usually the decoders are identical and are expressed by the same table. Sometimes, particularly for the counters, the combinational circuit that gives the next state can be eliminated because of an easier way available in the assembly language.

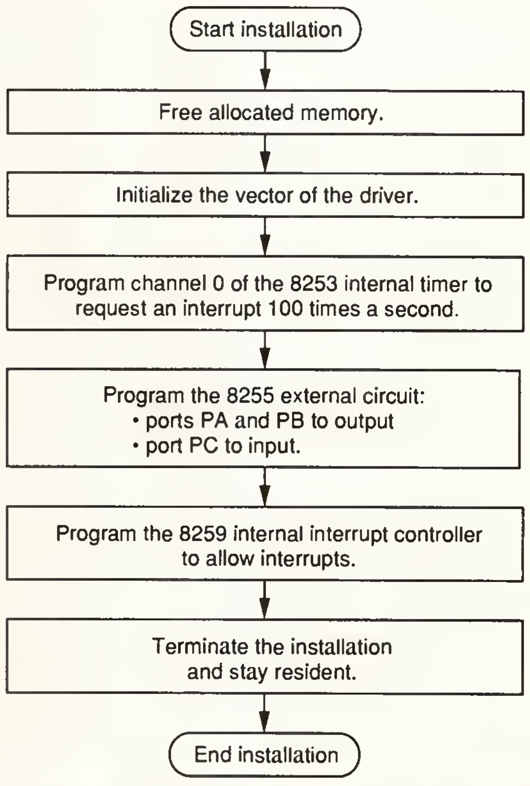

Start installation
Free allocated memory.
Initialize the vector of the driver.
Program channel 0 of the 8253 internal timer to request an interrupt 100 times a second.
Program the 8255 external circuit:
• ports PA and PB to output
• port PC to input.
Program the 8259 internal interrupt controller to allow interrupts.
Terminate the installation and stay resident.
End installation

Figure 7. The installation part of the driver.

Figure 9 gives the information displayed when the simulated structure represents the dice game described by a file in the database. In this case, the system generates two tables: one for the circuit that determines the next state and the other for the two identical decoders. The first part of the figure shows the dialogue between the user and the inference engine to identify the structure. The second part gives—for each multiple output combinational circuit that is a component of the structure, the input variables, the outputs, and the Boolean equations—information that was loaded from the database. The third part, common to each combinational circuit, lists the table of the product terms method, which the expert system generated and transferred to the driver. Note that, in the first table, the first byte indicates by the value 187 (6:6) the initial state of the dice game.

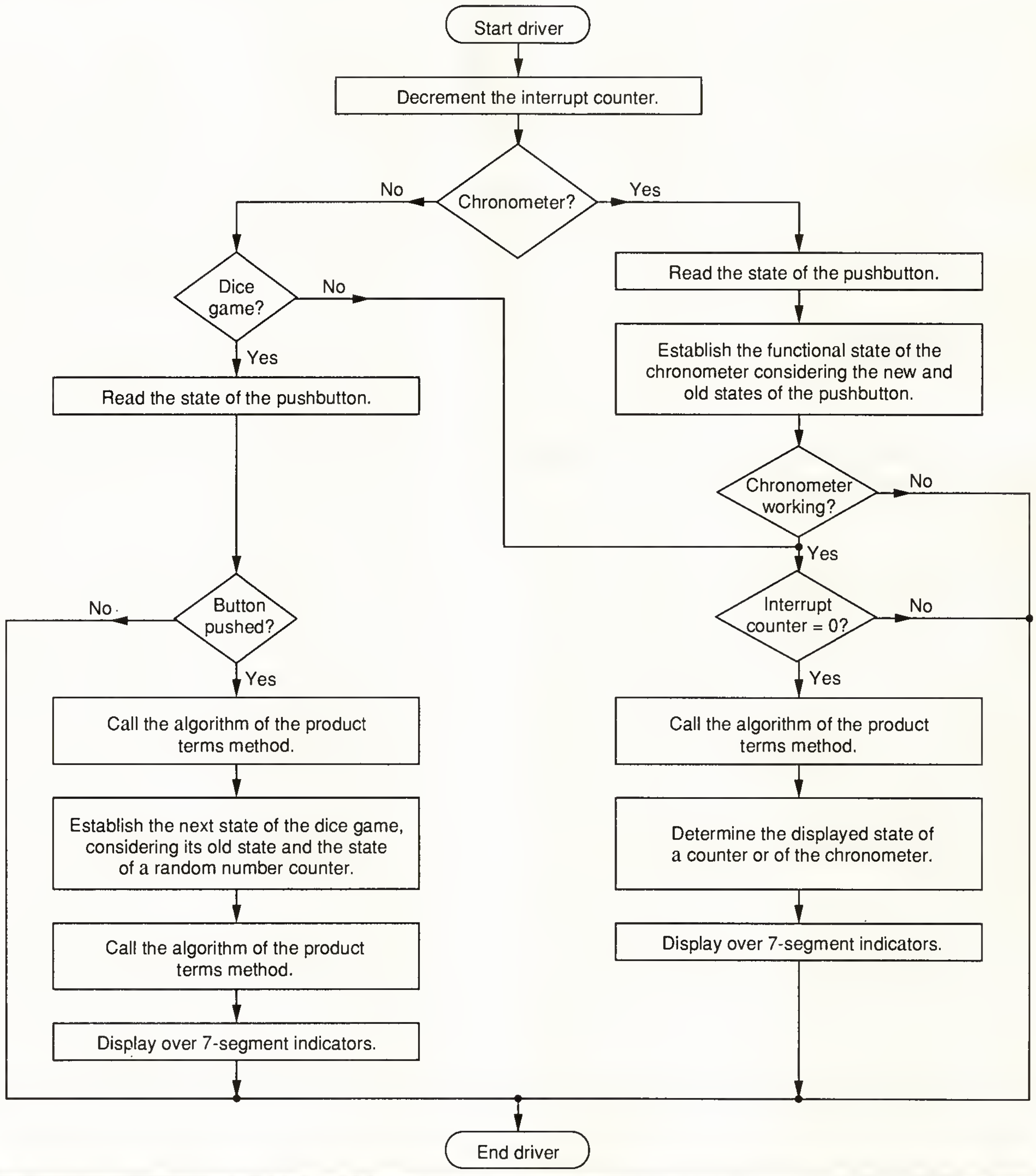

Start driver
Decrement the interrupt counter.
Chronometer?
No
Yes
Dice game?
No
Yes
Read the state of the pushbutton.
Read the state of the pushbutton.
Establish the functional state of the chronometer considering the new and old states of the pushbutton.
Chronometer working?
No
Yes
Button pushed?
No
Yes
Interrupt counter = 0?
No
Yes
Call the algorithm of the product terms method.
Establish the next state of the dice game, considering its old state and the state of a random number counter.
Call the algorithm of the product terms method.
Display over 7-segment indicators.
Call the algorithm of the product terms method.
Determine the displayed state of a counter or of the chronometer.
Display over 7-segment indicators.
End driver

Figure 8. The resident part of the driver.

Improved structures

As mentioned, once initialized, the driver works concurrently with other tasks to share the processor time. In our real interface example, we program the interface to request an interrupt 100 times/second.

To work correctly, we must analyze the driver to determine precisely the amount of time required to implement different structures. The time required to execute different branches of the resident part of the driver depends on several factors, among these are:

- the number of product terms that characterize every group of Boolean equations;
- the parameters introduced by the user in the dialogue with the expert system (for example, the value of a period for a counter); and
- the state of the different active elements of the driver (for example, the value memorized by the interrupt counter for the chronometer).

Therefore, we cannot make a general performance analysis; each logical structure must have its own analysis. If we consider two of our possible structures, the performance analysis gives:

- For the two-digit direct BCD counter with decoders, the average time required to implement the structure is 2.96 percent of the processor time. In any situation, the time consumed by the driver during one interrupt must be less than the time between two successive interrupts.
- For the dice game, the average time required to implement the structure is 4.56 percent of the processor time. The execution time of the driver routine can be greater than the time between two successive interrupts.

When the time analysis becomes critical, we propose two improved structures for practical applications of the product terms method with the expert system.

Figure 10 shows a situation in which *n* interfaces work concurrently to share the processor time. Each interface has its own driver, which receives the data from one and only one expert system. In this organization, another database stores the drivers, and the expert system loads each driver, after the identification of the structure. Consequently, one can write shorter drivers having shorter execution times. Of course,

```
Enter the logical equation ? (y/n) n

is it counter ? n

is it start_stop_clock ? n

is it game_of_chance ? y

It is a dice_game

Input variables wzyxqsrp

Output functions WZYXQSRP

W = p_w+px_y+px_w
Z = p_z+pz_
Y = p_y+px_z+px_w
X = p_x+px_yz
Q = p_r+p_q
S = s_
R = p_s+p_q
P = p_rs

187
129  128  128   49   33 128  145  129  160   65   64   64
 65    1   64   33   32  32   81   65   32   17   16   16
113   97   16    3    2   8    9    8   10    4    0    4
  5    4    2    7    6   1
  0

Input variables qsrp

Output functions ABCDEFG

A = sq+rq_
B = r_s+p_s_
C = rq+s
D = sq+rq_+p
E = rq_+p
F = rq+p
G = q+r

255
 12   12   72   10    2  76    6    4   32    5    0   32
 10   10   18    4    4  16    1    1   14    8    8    1
  2    2    1
  0
```

Figure 9. Information displayed in an example dice game.

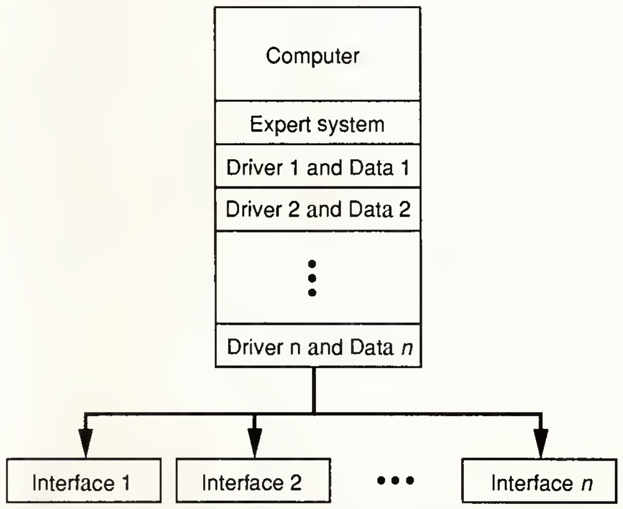


Figure 10. *N* concurrent interfaces loaded by an expert system.

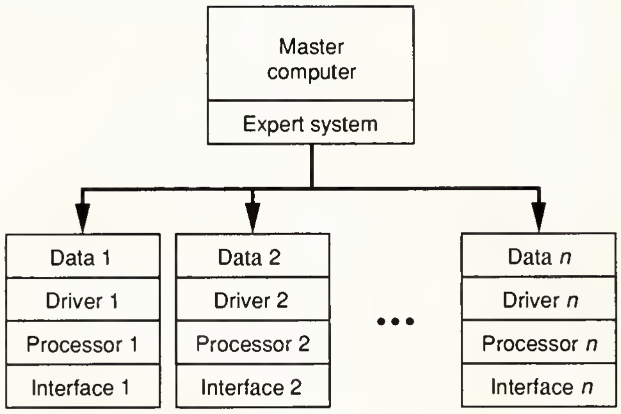


Figure 11. A master-slave structure loaded by an expert system.

this would always require a time/performance analysis, similar to the one done for a single interface.

Figure 11 presents another solution. Here, we use a master computer to manage the expert system and a number of very simple slave processors. We connect each slave processor to a physical interface so we can perform local computations. The expert system loads the drivers and the tables into the local memories. In this way, we obtain a parallel computation system in which the logical behavior of any interface could be changed dynamically.

Finally, if for some special applications, the sequential computation time became critical, we could develop a VLSI circuit. The recursivity of the algorithm presented in Figure 1 and the idea of parallel processing of product terms lead to the VLSI organization indicated in Figure 12. The basic cell of VLSI organization contains three one-word registers that store the mask word, the product word, and the output word of a logical product. The combinational part of the cell acts as a filter between the product term information and the outputs. The control part allows us to load the registers of the cells and consequently to change dynamically the logical behavior of the structure.

The physical structure of the VLSI circuit is regular, and we can consider it as we would a VLSI memory. Application would be similar to that of the recent VLSI family of programmable gate arrays developed by Xilinz.[11,12] Our proposed VLSI circuit is less flexible logically but, considering its simple cell architecture and regularity, it allows a greater density.

Our general method implements multiple output combinational and synchronous sequential circuits. We used the same implementation algorithm for all digital structures, each structure being characterized by one or several blocks of data. In other words, we developed a Boolean data processor. Software or hardware implements the proposed method.

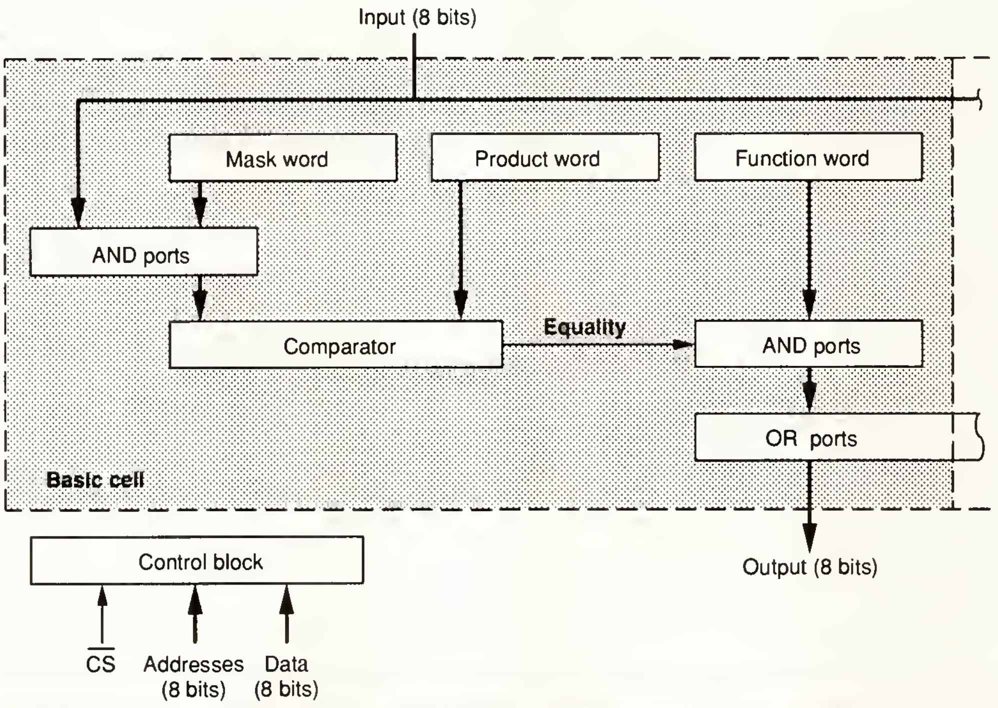


Figure 12. A proposed VLSI circuit for the product terms method.

The software applications area concerns programmable controllers and teaching hardware simulators or any other logical structure in which the speed of execution is not critical. For critical-time situations, we suggest a hardware VLSI circuit.

The important characteristics of the proposed method are:

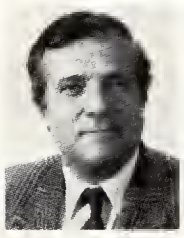
• the same hardware structure implements a large number of digital circuits,

• the user can change the behavior of the structure dynamically, and

• the classical expressions of functions into logical equations can be implemented directly by using the expert system.

References

1. Intel Corporation, *Microcontroller User's Manual,* Santa Clara, Calif., 1982.
2. E. Cerny, D. Mange, and E. Sanchez, "Synthesis of Minimal Binary Decision Trees," *IEEE Trans. Computers,* Vol. C-2, No. 7, July 1979, pp. 472-482.
3. P.J.A. Zsombor-Murray et al., ''Binary-Decision-Based Programmable Controllers, Part I, *IEEE Micro,* Aug. 1983, pp. 67-83.
4. P.J.A. Zsombor-Murray et al., "Binary-Decision-Based Programmable Controllers, Part II," *IEEE Micro,* Oct. 1983, pp. 16-37.
5. D. Telfer et al., "A Prototype Modular Sequence Controller with Binary Decision Architecture," *Proc. Mini and Microcomputers and their Applications,* 1986, pp. 105-109.
6. I.C. Dancea, "A Software Method for Implementation of Digital Circuits in Microcomputer Systems," *Proc. Software and Hardware Applications of Microcomputers,* 1986, pp. 145-148.
7. I.C. Dancea, "An Expert System to Generate and/or Modify the Logical Behavior of a Microcomputer Interface," *Proc. Computer Applications in Design, Simulation and Analysis,* 1988, pp. 148-152.
8. M.R. Dagenais, V.K. Agarwal, and N.C. Rumin, "McBoole: A New Procedure for Exact Logic Minimization," *IEEE Trans. Computer-Aided Design,* Vol. CAD-5, 1986, pp. 229-238.
9. H. Schildt, *Advanced Turbo Prolog,* Osborne McGraw-Hill, Berkeley, Calif., 1987, pp. 57-90.
10. Monolithic Memories, *Programmable Array Logic Handbook,* Santa Clara, Calif., 1983.
11. S.L. Landry, "Designer Logic and Symbols with Logic Cell Arrays," *IEEE Micro,* Feb. 1987, pp. 51-59.
12. Xilinx Inc., *The Programmable Gate Array Design Handbook,* San Jose, Calif., 1986.



Ioan Dancea is a professor in the Department of Computer Science at the University of Quebec in Hull, Canada, where he teaches computer architecture, parallel processing, and computer simulation. He also held this position on the Faculty of Electrical Engineering at the Polytechnic Institute of Cluj in Romania, where he taught computer architecture and digital systems. Mainly interested in multiprocessor systems, parallel languages, and software implementaton of digital circuits, he has authored several Romanian and French computer books.

Dancea received the diploma of electrical engineer and the PhD degree in computer engineering, both from the Polytechnic Institute of Bucharest in Romania.

Questions concerning this article may be addressed to the author at the Department of Computer Science, Universite du Quebec a Hull, Case Postale 1250, succursale "B," Hull, Quebec J8X 3X7, Canada.

Reader Interest Survey

Indicate your interest in this article by circling the appropriate number on the Reader Interest Card.

Low 156 **Medium** 157 **High** 158

FEATURE

An Extensible DBMS for Small and Medium Systems

This flexible database manager queries and updates relations implemented as combinations of data and B⁺ tree files in Unix and other C environments.

Mike P. Papazoglou

GMD (German Research Center for Computer Science)

Today's inexpensive, powerful hardware encouraged designers to develop small- and medium-scale database systems. The commercial demand also heavily engaged software companies and educational institutes in the design of database management systems. DBMSs, though in general possessing some rather stringent requirements with their functionality, must meet the diverse needs of the business community and at the same time be easy to use. For example, a large number of users need an interface that permits them to express their problems in a natural way.

These points imply that database systems should offer users a powerful, yet simple, data model to use as the basis for the specification of their conceptual requirements. Data-independent systems based on the relational model provide such facilities. Relational systems give users a substantial degree of flexibility and therefore help them cope with the demands imposed by a wide spectrum of applications.

Ideally, a DBMS that makes users feel comfortable lets users perceive the way the query language allows them to formulate their requests as adequate and satisfactory. In this sense, it would be highly desirable to have a choice of open-ended query interfaces and a means of easily providing some novel ones. Thus, DBMSs offering a set of modules in the form of database function specifications that permit the development of customized query language interfaces to the system are highly desirable.

We call such DBMSs *extensible.* An extensible DBMS hopefully offers both a powerful and an efficient system kernel that allows for extensions by customizing its basic functions on the uppermost level. Designers should work to make such an extensible DBMS "query language independent." An independent approach provides mechanisms for language designers to design experimental query languages without the need for the intricate file access and manipulation code that constitutes the file manager component of the DBMS.

I describe an extensible, relational database management system called Requiem. Requiem meets the above-mentioned needs and offers users a truly relational interface. This latter feature stresses the fact that Requiem differs from most of the contemporary commercial or experimental products that claim to be relational although they fail to meet main relational qualification requirements.[1]

In general, small or medium DBMSs fail to meet serious relational qualification requirements either by not supporting integrity mechanisms,

0272-1732/89/0400-0052$01.00 © 1989 IEEE

by not being transparent as far as indexing is concerned, or by not offering view and catalog manipulation facilities. The literature[1-3] extensively discusses the advantages of relational systems meeting the above requirements so that they hardly require any further elaboration or commenting.

Requiem (for RElational Query and Update Interactive SystEM) is an interactive system with facilities for querying and updating a database. A relational database system implemented on top of the Unix operating system, Requiem currently runs on a Sun computer under operating system 3.4. It also runs on the Apple Macintosh and the IBM PC. We developed Requiem primarily as a prototype of a single-user system to demonstrate the feasibility of supporting a full-fledged relational query language called RQL. However, the system provides an open-ended structure in that it can easily be extended to incorporate multiuser mechanisms.

Requiem offers a complete query language interpreter, which consists of a set of special-purpose modules written in C. The interpreter provides its own tools for lexicographical analysis and parsing. Requiem's parser employs incremental parsing techniques and is a highly complex module that incorporates a complete predicate expression compiler and consists of roughly 2,300 lines of C source code. The lexical analyzer, also a highly specialized tool, performs, among other jobs, view checking and expansion; it comprises 750 lines of C code.

The entire Requiem implementation consists of just over 10,000 lines of code and occupies 140 Kbytes on disk. Requiem requires roughly 150 milliseconds (CPU elapsed time) to perform rather complex retrieval operations involving indexing and hashing on a relation consisting of 500 records with 10 fields of data. We measured this performance estimate on a Sun 3/140 by employing normal Unix resource utilization routines. The concise system architecture, the rationale behind the entire system implementation, as well as the entire source-code listings for Requiem can be found in Papazoglou and Valder.[3]

The fact that Requiem does not rely on the assistance of such language development tools as *yacc*, or *lex*[4] implies that Requiem and RQL can, with reasonable effort, be ported to non-Unix environments supporting C. For example, we successfully ported Requiem to the Apple Macintosh (using the Lightspeed C compiler) and the IBM PC (using the Microsoft C compiler). My ambition is to provide users with a DBMS and a query language portable to a variety of small to medium systems.

Requiem supports the RQL query language with all its powerful facilities. However, at the same time Requiem provides an essentially query language-independent interface, in that it actually executes low-level commands. This capability lets users experiment with new query language structures that can interface with it with minimal effort. In essence Requiem provides a mechanism for language designers and system programmers to design experimental query languages without having to worry about the intricacies of file access, optimal access paths, and manipulation of codes being catered to by the file manager.

The Requiem organization

In Requiem a set of definitions expressed by means of a special RQL sublanguage called the *Data Definition Language* specifies the database schema. DDL, a descriptive language, allows users to describe and name the entities required for applications and the associations that may exist between the different entity descriptions. We use DDL to develop a schema or even to modify an existing one. It cannot be used to manipulate data.

The interpretation of the DDL statements produces a set of records stored in special system-defined tables collectively referred to as *data catalogs*. Data catalogs contain metadata, that is, data that specifies data. They contain definitions of records, data items, and other objects that are of interest to users or are required by the DBMS. The DBMS normally consults the data catalogs before modifying the actual data in the database system.

We collectively refer to all the software modules of Requiem that deal with the implementation of DDL operations as the *DDL Processor*. A functional description of the DDL Processor appears in Figure 1 on the next page. This figure shows how the various software components of the DDL Processor interact with each other to implement a user's DDL request. First, it checks each request to conform to the predefined lexical and syntactical constructs of the DDL, and then it calls the appropriate DDL functions. These functions either create a new database schema or update an already existing one. Consequently, relations and attributes are included in the final database schema or deleted from it, as users wish.

The Requiem *Data Manipulation Language* provides a set of operations that support the basic data manipulation operations on the contents of the database. Data manipulation operations usually include:

• the insertion of new data into the database,
• the modification (updating) of data stored in the database,
• the retrieval of data contained in the database (query language), and
• the deletion of data from the database.

Thus, as one of its primary functions, Requiem supports a data manipulation language in which users can formulate commands that will cause such data manipulation to occur. In this context we use the terms *data manipulation* and *query language* interchangeably for

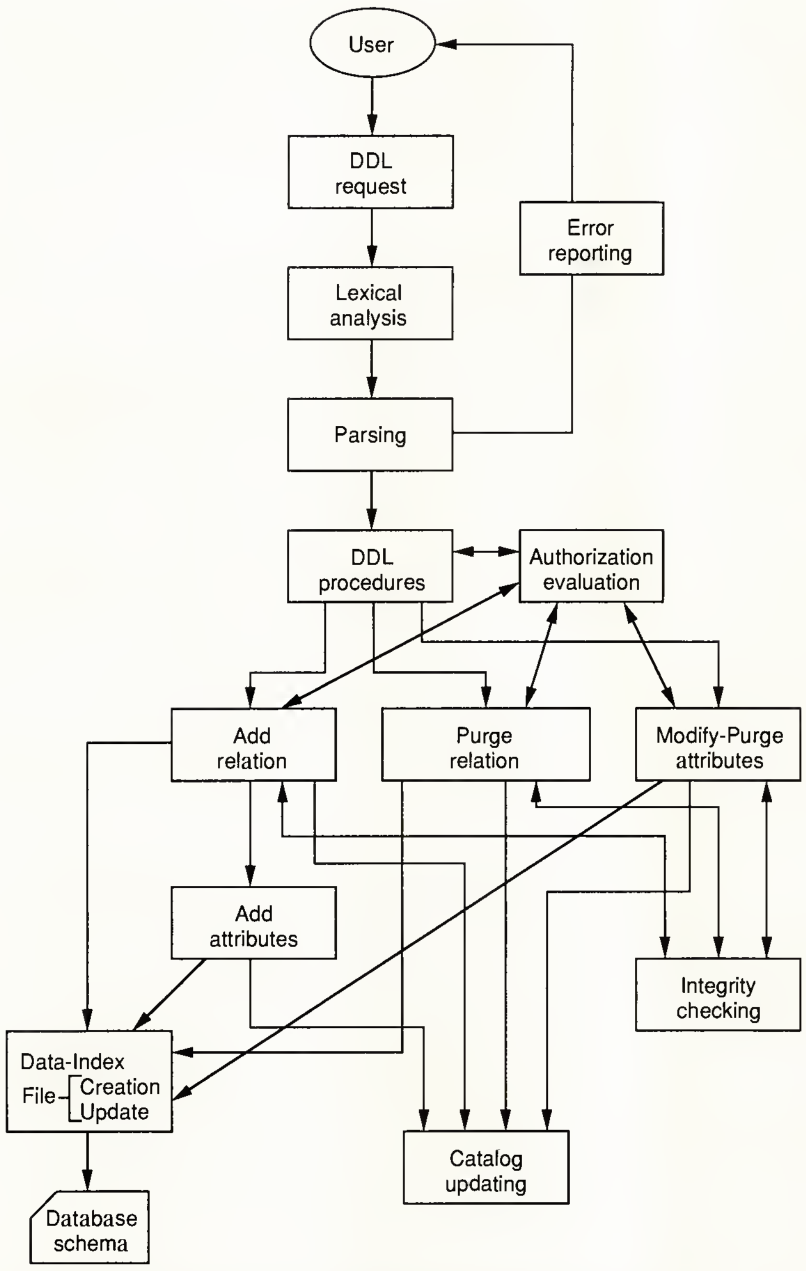


Figure 1. A functional description of the DDL Processor.

reasons of simplicity, although this is technically incorrect.

We collectively refer to the part of Requiem that deals with the implementation of DML operations as the *Query Processor.* See Figure 2 for a functional description of Requiem's DML Processor. This figure shows how the various software components of the Query Processor interact with each other to implement a user's DML request. The processor first checks each DML request (query) to see that it conforms to the predefined lexical and syntactical constructs of the DML, then it calls the appropriate DML functions.

These functions decompose each query into its constituent parts. When a predicate is involved in a query, the functions first check the predicate for syntactic correctness and subsequently generate the appropriate environment (abstract stack machine) for query evaluation. Each tuple, or table row, is evaluated in turn, and all the tuples satisfying the predicate finally move to a transient relation file, which is the outcome of the posed query. All tuples are also checked for semantic congruence at runtime; users receive reports of any resulting errors, bringing the evaluation to an immediate end.

Users normally interact with Requiem on line by issuing various query language statements supported by this DBMS. The query language interpreter processes these statements and calls the appropriate DBMS procedures to perform the requested operations.

In addition to providing a query language, Requiem like most modern DBMSs, offers users the option to write application programs in a general-purpose programming language. These application programs use predefined Requiem DML procedures that achieve communication with the DBMS and consequently with the database contents. To users, the DML procedures appear to be merely an extension of the C programming language. Most of these DML procedures are exactly the same as the ones supporting the query language; however, some additional DML procedures eventually have to be mapped into C statements that call the standard DBMS procedures. The modified source program is then compiled in the conventional way (see Figure 3). Thus, Requiem does not support an *embedded sublanguage*; it rather provides users with a full *programmable procedural interface.* Accordingly, we use predefined C procedures to write all customized user interfaces.

Each of the above approaches has its own merits. With a query language, users obtain results much more quickly because there is no need to write and debug programs. However, the query language contains built-in limitations, as it quite difficult to perform functions for which the language was not defined. Note also that although a DML programming interface allows users to acquire all the power and the flexibility offered by the general-purpose programming language, it requires much more effort and maturity from users.

Some systems do not support a procedural interface; they provide an embedded sublanguage that requires the services of a preprocessor. Thus the label procedural interface depicted in Figure 3 should be replaced by that of a preprocessor. Ingres,[5] which supports an embedded sublanguage called Equel, supplies more details and examples concerning this issue.

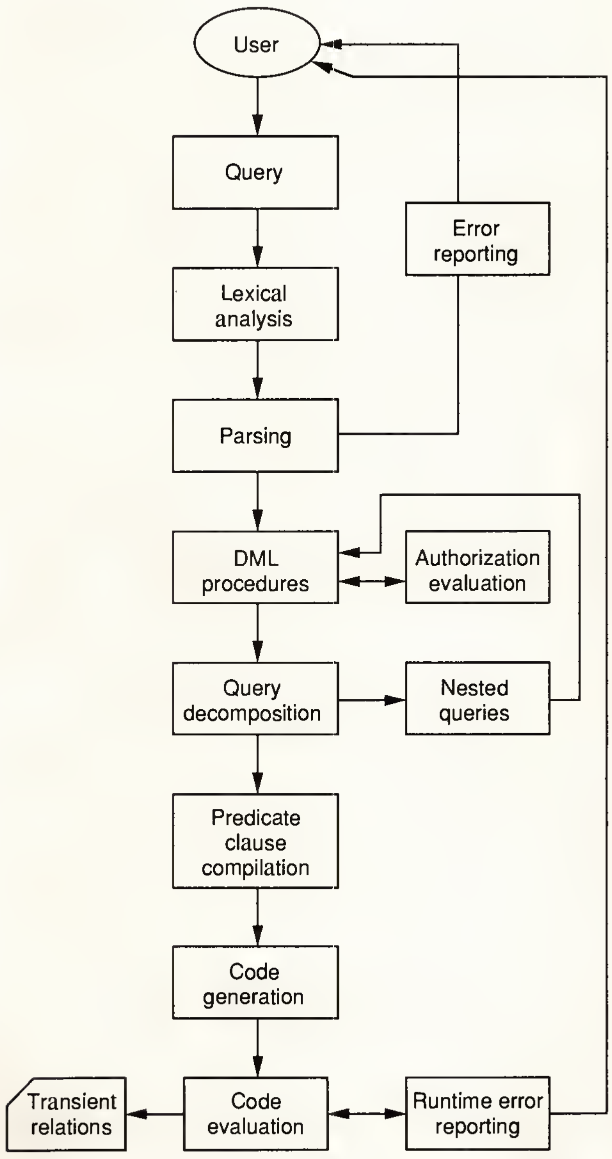


Figure 2. A functional description of the Query Processor.

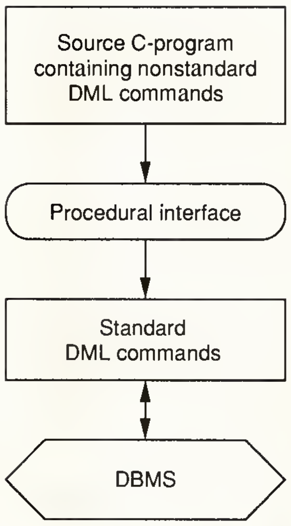


Figure 3. Interaction with the DBMS via nonstandard DML procedures.

The Structure of RQL

In designing RQL, my philosophy has always converged into two principal design issues:

• **Designers of query languages should always be parsimonious in the number and complexity of the constructs they introduce into the language.** A powerful language with many and probably complex constructs does not necessarily imply its acceptance or appreciation by a wide community of users. The goal should be to introduce fewer constructs with general applicability and less complexity. This goal is probably easier to implement and at the same time guarantees that queries formulated in that particular language are probably easier to consider logically.

• **Designers should provide clear user-interfaces so that the syntax and the structure of the language can be altered or augmented instantly at any time to reflect the potential needs of users.** Such interface facilities can be constructed as front-ends to an existing DBMS and improve the process of identifying and retrieving required information from the database. For example, one could provide a friendly user interface facility for the development of query languages for nonprogrammers. Furthermore, this goal implies that an extensible query language may be preferable, for practical purposes, to a more powerful language offering a rigid structure.

RQL has points in common with SQL, or Structured Query Language, (actually in some cases the RQL syntax resembles that of SQL) in the sense that it comprises a complete stand-alone query language. This capability frees users from such major decisions and concerns as how the data structures are implemented in the internal level and what kind of algorithms are operating on the data stored in the database. Moreover, RQL provides users with the notions of physical and logical data independence, as well as relational data integrity.[6]

The following outline of RQL basically concentrates on the functional capabilities of RQL and provides several examples to clarify our points of view. The adjoining box contains a complete description of the language syntax.

Relation implementation and user perspective. The principal elements of RQL are *relations* (in the form of tables) that are physically stored in secondary memory. RQL represents each nonderived relation in Requiem as a combination of a stored data file and an index file. Each data file comprises a file header and a file body. The file header contains the definition of the relation including names, types, and sizes of all the attributes contained in the relation, the key definitions, the size of tuples, and so on. The remainder of the file contains a number of fixed-size records representing the tuples of the relation.

RQL stores each record as a byte string of fixed length, which corresponds to the maximum length of a tuple. On the other hand, Requiem supports multiple indexes in an index file in which the indexed attribute values may vary in length from one index to another. Indexing techniques internally handle duplicate keys of any data type, multiple entries pointing to the same record in a stored data file, multiple indexes in use, variable length keys, and sequential advance through an index.

When a relation is created, RQL encrypts its name, and the index and data files of this relation appear simultaneously in the user's current directory. Users issue statements to create indexes either explicitly or implicitly and can also explicitly destroy them, but only if they possess that privilege. Users do not have to state where and how a certain index

The RQL Syntax

Data definition statements

```
<OOL-etatement>   :: = CREATE <rel-name>  ( <attr-def> { , <attr-def> }
                                        [, <ref-clauee> <ref-clause> ] );
                   |   OEFINE <view-name> AS <rql-query>
                   |   OROP   <view-name>
                   |   MODIFY <rel-name> <interactive>
<attr-def>        :: =   <attr-name> ( <attr-type> ( <integer> ) [ , <key-spec>] )
<attr-type>       :: =   CHAR | NUM | REAL
<key-spec>        :: =     UNIQUE | SECONOARY
<ref-clause>      :: = FOREIGN KEY <attr-name> REFERERENCES <attr-name>
                                                      IN <rel-name>;
<privilege-def>   :: = GRANT <privileges> ON <rel-name> TO <grantee>;
                   |   REVOKE <privileges>  [ ON <rel-name> ] FROM <grantee>;
<privilegee>      :: = DBA | <op-liet>;
<op-liet>         :: = SELECT | INSERT | JOIN | PROJECT
                   |   EXPOSE | GROUP | UPOATE | OELETE
<grantee>         :: = DBA | <ueer> { , <ueer> }
<user>            :: = <ueer-id> | GROUP <group-name>
```

Data manipulation statements

```
<OML-statement>   :: =    <query-term> ;
                   |   EXPOSE  <retrieve-clauee> <expoee-clause> ;
                   |   OELETE  <del-clause> ;
                   |   INSERT  <rel-name> <interactive>
                   |   UPOATE  <select-clause> <interactive>
<query-term>      :: =   OIFFERENCE  <rel-name>  ,  <rel-name>
                   |     INTERSECT   <rel-name>  ,  <rel-name>
                   |     JOIN     <rel-name>  ON  <attr-name>   WITH
                                <rel-name>  ON  <attr-name>
                   |     PROJECT  <retrieve-clause> [<group-clause>]
                   |     SELECT   <eelect-clause>
                   |     UNION    <rel-name>  ,  <rel-name>
<select-clause>   :: =  <sel-attre> [ <from-clause> ] [ WHERE <pred-clause> ]

<retrieve-clauee> :: =  <rel-clauee> [  over  ] <sel-attrs>
                                        [ WHERE <pred-clause> |<in-predicate> ]
<rel-clause>      :: =  <eel-rele> | <nesting-statement>
<from-clause>     :: =  FROM  <sel-rele>
                   |    <eubquery-term>
<eel-rele>        :: =  <eel-rel>  ,  <eel-rels>
<sel-rel>         :: =  <rel-name> [<alias>]
<sel-attrs>       :: =  ALL
                   |    ( <eel-attr>  {, <eel-attr> } )
<sel-attr>        :: =  <attr-name> [<aliae>]
<subquery-term>   :: =  ( <query-term> )
<group-clause>    :: =  GROUP BY  <attr-name> <aggr-clause>
<aggr-clause>     :: =  WHERE  <aggr-func> <aggr-attr> <rel-op> <scalar>
                   |    <aggr-func> <aggr-attr>
<aggr-attr>       :: =  ( <attr-name> )
<aggr-func>       :: =  AVG  |  COUNT  |  MAXM  |  MINM  |  SUM
<del-clause>      :: = <view-name>
                   |   <retrieve-clauee>
<expoee-clause>   :: = WHEN  <attr-name>  IS  IN
                       PROJECT  <rel-name>  OVER  <attr-name> [ <pred-clause> ]
```

Auxiliary statements

```
<aux-statement>   :: = ASSIGN   <rel-name>  TO  <query> ;
                  |  COUNT    <rel-name> ;
                  |  DISPLAY  [<signed-no> [, <integer> ] ]
                                              USING <retrieve-clause> ;
                  |  EXIT;
                  |  EXPORT     <rel-name>  INTO  <file-name>;
                  |  EXTRACT    <rel-name>  INTO  <file-name>;
                  |  FOCUS  ON  <rel-name>;
                  |  HELP;
                  |  IMPORT  <file-name>  INTO  <rel-name>;
                  |  PRINT   <print-clause>  [  INTO  <file-name> ];
                  |  QUIT;
                  |  SHOW  VIEW   <view-name>;
<print-clause>    :: =  <using-clause> <select-clause>
                  |     <retrieve-clause>
<using-clause>    ::=   USING <file-name>
<file-name>       ::=   <identifier>
```

Predicate expression statements

```
<pred-clause>     ::=  <attr-name> <rel-op> <expression>
<in-predicate>    ::=  <attr-name> [NOT]  IN  <in-list>
<in-list>         ::=  ( <scalar> { , <scalar> } )
<expression>      ::=  <primary> { <logop> <primary> }
<primary>         ::=  <statement> | <assgnt> <relop> <simple-expr>
<statement>       ::=  <simple-expr> { <relop> <simple-expr> }
<assignment>      ::=  ( <variable> := <simple-expr> )
<simple-expr>     ::=  <term> { <addop> <term> }
<term>            ::=  <factor> { <mulop> <factor> }
<factor>          ::= <operand>
                   |  ( <expression> )
                   |  <nop> <factor>
<operand>         ::=  <integer> | <real> | <string> | <attribute> | <variable>
<variable>        ::=  $<chars>$
<attribute>       ::=  [<rname>.] <aname>
<integer>         ::=  <digit> [ <integer> ]
<real>            ::=  <integer>.<integer>
<signed-no>       ::=  <addop><integer>
<string>          ::=  ',,<chars> "
<chars>           ::= nil | <chars> { <character> }
<nop>             ::= -
<logop>           ::= & | |
<relop>           ::= = | <> | < | > | <= | >= | ==
<addop>           ::= + | -
<mulop>           ::= * | / | %
<digit>           ::= 0 | ... | 9
```

Miscellaneous

```
<view-name>       ::=  <identifier>
<alias>           ::=  <identifier>
<rel-name>        ::=  <identifier>
<attr-name>       ::=  <rel-name>  .  <identifier>
                  |    <identifier>
```

should be used. The decision as to whether a specific index is going to be used in response to a query is entirely dependent upon the system. Moreover, RQL data manipulation statements never explicitly mention indexes.

In Requiem users can create relations that fall into two major categories:

- base relations, and
- views.

A *base relation* is a "concrete" relation in the sense that it physically exists in terms of physically stored records and indexes. In contrast, a *view* is a virtual or derived relation in the sense that it does not actually physically exist in storage, but that users perceive it as a real table. Users define views, as will be explained later, in terms of one or more underlying base relations.

Internal system organization and access methods. The file header of a base relation includes information pertinent to the relation. Such information includes the names, types, and sizes of attributes, the total number of attributes contained in the base relation, key definitions, and so on. Therefore, at any instant we can locate any requested attribute value.

Although this storage scheme consumes more space than normally required, it is faster than that of variable-length records because updates in storage schemes of variable-size records may require major, time-consuming file rearrangements. Fixed-length records are certainly much simpler to implement and manipulate. Furthermore, the file manager does not have to save tuple identifiers including indications of the tuple length or even of individual field lengths.

In the Requiem fixed-record storage scheme, the space freed by a deleted record in the data file duplicates exactly that of the space required for a subsequent record insertion. This space can be reclaimed for further use without having to rely on such time-consuming techniques as record compression. Actually, Requiem uses an available record list that consists solely of a set of deleted, chained-together records. Once this list is exhausted, records can start being inserted at the bottom of the data file.

Internally, Requiem organizes indexes as *B⁺ trees,* with each block of the index occupying one page in storage. To speed up operation, Requiem uses a *cache buffering* scheme to reduce the number of actual disk reads. In this context cache buffering denotes a collection of blocks that normally belong on the disk but are also being kept in memory space to improve performance. We kept the algorithm used to manage the cache simple and based it on LRU (least recently used) techniques.[7,8]

Blocks residing in the current index position are likely to be referenced again so that they become candidates for caching. Any updates or deletions occur simultaneously on the contents of the cache blocks and on corresponding index file blocks held in disk. In this way information in the cache buffer always remains consistent with the corresponding index blocks in the file. The Requiem cache method is also quite simple; it allocates space for one block at each index level and keeps the current block in the cache. Cache buffers are allocated in such a way that every open index file obtains its own set of cache buffers.

The database structure

In Requiem a database can be viewed as a collection of tables (relations), each of which holds a unique name. These tables constitute the logical and certainly not the physical data structure part of a relational database. This fact implies that at the internal level the system can freely use any of the traditional techniques (sequential files, indexing, inverted files) provided that it is capable of mapping those structures into table constructs at the logical level. Each table consists of a *table header* and a *table body*. The table header consists of a fixed set of attribute names, and the body of a table consists of a time-varying set of rows called tuples. The tuples are distinct from one another and their total number varies with time. To put the whole idea into perspective, consider the *supplier* relation of the suppliers-and-products database as depicted in Table 1.

The header of the relation supplier consists of the three attributes Sup_id, Sup_name, and Location. The values of Sup_id come from the domain of supplier identifications, the values of Sup_name from the domain of supplier names, and finally the values of Location from the domain of supplier locations. The body of the relation supplier consists momentarily of four tuples, and each of these tuples consists of three <attribute – name: attribute – value> pairs. Exactly one such pair corresponds to each of the three attributes contained in the header of supplier. Consider the supplier tuple with the identification number 605:

Table 1. The suppliers-and-products database.

Supplier

Sup_id	Sup_name	Location
605	Jones	London
612	Schmidt	Zurich
827	Dupont	Paris
855	Dick	Bonn

Product

Prod_no	Prod_name	Weight	Finish
10	Desk	150	Pine
12	Bookcase	100	Cherry
21	Cabinet	80	Oak
25	Sofa	20	Oak
33	Dresser	90	Pine

Supply

Sup_id	Prod_no	Quantity
605	21	150
612	10	200
827	12	120
855	25	135
605	33	100

<SUP_ID: 605>, <SUP_NAME: Jones>, <LOCATION: London>

From what has been mentioned thus far, one can certainly understand that the smallest unit of data supported by the relational data model is that of the individual data value. Such data values are *atomic* in the sense that they cannot be further decomposed as far as the model is concerned. A set of data values all having the same *type* can be drawn from a certain reservoir of data values that actually comprises their corresponding domain.

In retrospect, here are some relation properties of vast importance to the material that follows:

• *Individual data values are atomic.* Attribute values cannot be decomposed, that is, when any arbitrary numerical or character string value is decomposed into smaller fractions, it changes its substance, and hence loses its meaning.

• *Values corresponding to distinct attributes are homogeneous in nature.* This property implies simply that all attribute values belonging to a specific domain are all of the same type.

• *All tuples in a relation are distinct.* A relation should never contain identical tuples.

• *The order of attribute names in a relation header is immaterial.*

Data definition functions. Data definition in RQL allows users to create base relations. DBMSs very rarely explicitl, employ, if at all, the notion of domain as supported by the relational calculus. Consequently, Requiem defines relations in terms of their attributes and the notion of domain appears only implicitly through the types (integer, real, or character) that the values of the attributes are assumed to take on. Each RQL attribute can be declared with one of the following three data types:

• integer,
• real, and
• character string.

Attribute values must always be defined in a base relation, a fact which provides a minimum safeguard against certain inconsistencies. Furthermore, no keys are explicitly declared in the relation scheme, as the notions of primary, foreign, and secondary keys are inherent in RQL at the access path level. Requiem defines indexes by using the Unique statement during the process of relation creation. The option Unique allows users to specify that an attribute is a discriminating key (primary key) for a given relation. Furthermore, it indicates direct tuple access paths having a given key value. The data definition statements of RQL, which internally support key definitions, conform to the relational data integrity rules as stated in Date, Volume 2.[6] Notice that users cannot normally explicitly use Unique when creating indexes, as for example in SQL, since all unique indexes must be created during the relation definition process. All other indexes explicitly created by users via the Secondary statement result in the introduction of *secondary keys.* Secondary keys permit quick and direct accessing of tuples in a relation data file.

The Reference clause introduces *foreign keys,* which act as intermediaries to represent references from one relation to another. The combination of any two foreign key attributes taken together is a *primary key* for the underlying relation. In fact in RQL, users explicitly specify referential integrity constraints. Actually, users specify the referential constraints declaratively as part of the schema definition; the system itself then imposes and maintains these constraints.

To define a conceptual schema for the suppliers-and-products database as depicted in Table 1, we should use the Create <relation – name> statement as follows:

```
create SUPPLIER (sup_id (num(5), UNIQUE),
                 sup_name (char(20), SECONDARY),
                 location (char(20)));

create PRODUCT (prod_no (num(5), UNIQUE),
                prod_name(char(20),SECONDARY),
                weight (real(10)),
                finish (char(15)));

create SUPPLY (sid (num(5)),
               prod_no (num(5)),
               quantity (num(10)),
               foreign key sid references
               sup_id in SUPPLIER;
               foreign key prod_no references
               prod_no in PRODUCT;);
```

As a result of the above statements suitable entries will be made in the data catalogs, and space will be allocated for the base relations. Each new entry describes a new empty relation for the time being.

Some interesting points in the foregoing example require further elaboration:

• The relations supplier and product acquire two primary keys, SUP_ID and PROD_NO, as a result of the Unique declarations.

• The relation supply contains two foreign keys, namely SID and PROD_NO, whose combination actually forms the primary key of the relation. This comes as a direct consequence of the Unique declarations in the relations supplier and product and the Reference clause that immediately follows the definition of relation supply.

Notice the absence of Unique attribute declaration in the relation supply. You must realize the fact that supply is not a stand-alone (independent) relation; rather it associates the two formerly defined relations with one another. We call such relations that cannot exist independently *composite* relations. Composite relations in RQL require that their referenced *target* relations be defined prior to their own definition. Finally, one word of caution: Requiem requires that a composite relation can contain only two foreign keys whose combination is automatically interpreted as the primary key for this specific relation.

• The relations supplier and product declare their attributes Sup_name and Prod_name to be indexed to speed up any retrieval operations based on these attributes.

Requiem requires every relation to have one distinguishable primary key. The requirement contrasts with some proposals that suggest that relations should be permitted to include any number of effectively interchangeable unique keys. As seen previously, Requiem provides a general integrity support mechanism along the lines of the Create statement for primary and foreign key support and follows Date's position that every relation should be allowed exactly one principal key, namely its primary key.[6] During the process of relation creation and data (attribute) definition, Requiem converts the entire range of definitional information from the logical structure to storage structure and media allocation. Any updates or deletions of primary or foreign key values are performed along the lines of the referential integrity rules specified by Date.[6]

The statement Modify <relation – name> modifies the names and lengths of the attributes of a particular relation. All attribute names and lengths are modified except for those of primary keys. The Modify statement actually commences a dialogue with the DBMS as to which attribute names and/or lengths should be altered.

Data manipulation functions. Database manipulation in RQL specifies any operations made against the contents of the database. Such operations include database interrogation and update. The data manipulation operations in RQL fall into four major categories: insertion, retrieval, modification, and deletion. The data manipulation part of RQL operates on both base and derived relations (views). At this stage we will concern ourselves only with base relations; furthermore we assume that queries are entered at and results are displayed on an on-line terminal.

Data insertion. The statement Insert <relation – name> inserts new tuples into a relation. Here users type in the name of the relation after the statement Insert, and Requiem answers with the name of the first attribute. Consequently, users are prompted for the values of attributes to be inserted. During the process of data insertion, Requiem prescribes the format for input source data, and this provides a minimum safeguard against data inconsistency. The underlying data file is subsequently specially prepared to accept the data and store it into the database.

The mapping technique used to transform the input data before storage in the database is quite simple. All input data should first be validated and then converted into byte strings (character strings). Validation of the data to be input is made on the basis of information stored during the relation definition process and of supplementary information provided by the system catalogs. Any data not conforming to the validation criteria are immediately rejected, and users are subsequently prompted to supply valid input data.

Data retrieval. During the process of data retrieval, users write a series of queries that generate an intermediate flat file as output. The flat file resulting from a retrieval statement can either be displayed on the terminal by issuing the statement Print, which causes intermediate file flushing, or can be fed directly to another retrieval operation as an operand.

The selection process identifies a subset of the tuples contained in a relation. The inputs to the selection operator are a named relation file and a predicate expression, which provides the criteria for selecting the tuples in the specified relation. The predicate selection expression should contain at least the name of an attribute derived from the header of the relation as a variable operand. In RQL the predicate selection expression is in fact a logical expression, which yields a Boolean result indicating which tuples should be selected and which should be rejected. The output of a selection statement is a flat file containing a subset of the tuples held in the original file, and some communication message to users concerning the anticipated volume of output tuples. In fact, every retrieval statement produces analogous communication messages as output. The output file could be a virtual file in the sense that it sometimes does not physically materialize although it is logically defined. This issue is obviously both entirely system dependent and user transparent.

Table 2.
Result of a simple selection query.

Prod_no	Prod_name	Weight	Finish
21	Cabinet	80	Oak
25	Sofa	20	Oak

Let us consider some simple sample queries using the suppliers-and-products database of Table 1. To select all those tuples from the product relation in which the products have an oak finish, we should use the following Requiem expression:

```
select all
   from PRODUCT
        where finish = "oak";
```

Table 2 illustrates the result of the previous selection query.

We may combine any valid logical expression in the predicate of a selection expression. For example, to select all those tuples from the product relation where the products have an oak finish and a weight heavier than 50, one should use the following RQL statement:

```
select all
   from PRODUCT
        where finish = "oak" & weight > 50;
```

It must be mentioned that the operation Select selects a number of tuples as specified by the Where clause; the keyword All is specified to indicate that complete tuples must be selected. Moreover, RQL uses the == partial comparison operator to determine whether it should accept an entry or not. In fact, this is a practical requirement because users may, for example, not remember the full name of a supplier or a product. Consider the following query:

```
select all
   from SUPPLIER
         where sup_name == "Ja";
```

This query produces as a result all Supplier tuples that include a Sup_name attribute value that starts with the two letters *Ja*.

In RQL the unary relational operation project allows columns in a relation to be masked. The projection operation yields a vertical subset of a given relation; that is, it yields that subset obtained by masking specified attributes. In RQL the attributes to be projected should be separated by commas and follow the Over clause. Any duplicate tuples resulting from a project operation are eliminated. For example, to obtain a relation containing product names and their corresponding weights, one should use the following RQL expression:

```
project PRODUCT
        over prod_no, weight;
```

The result of this query appears in Table 3.

On the other hand the query "Find the names and weights of these products which have an oak finish" can be specified as follows:

```
project
    (select all
         from PRODUCT where finish =
              "oak" & weight > 50) over prod_no, weight;
```

Table 4 depicts the result of this query.

To select data from multiple semantically related tables, users must use the Join construct. Let us consider the query "Find for all suppliers the products that they supply and the quantities they have in stock." This query is expressed as follows:

```
join SUPPLIER on sup_id
        with SUPPLY on sup_id;
```

The term *Join* arises from the fact that the relations supplier and supply are literally joined on their common attribute Sup_id. Observe that one of the two common attributes has been eliminated in the final result. Such a Join operator is normally called the *natural join*[1] and is the only Join operator supported in RQL. The result of this query is a relation as shown in Table 5.

Table 3.
Result of a simple projection operation.

Prod_name	Weight
Desk	50
Bookcase	100
Cabinet	80
Sofa	20
Dresser	90

Table 4.
Result of the nested query.

Prod_name	Weight
Cabinet	80
Sofa	20

Table 5.
Result of the unconditional Join operation.

Sup_id	Sup_name	Location	Prod_no	Quantity
605	Jones	London	21	150
605	Jones	London	33	100
612	Schmidt	Zurich	10	200
827	Dupont	Paris	12	120
855	Dick	Bonn	25	135

A Join operation in RQL can be conditional if it is combined with a predicate. Thus, we can formulate the query "Find for all suppliers the products that they supply and the quantities they have in stock provided that they supply more than 130 items of any kind" as follows:

```
join SUPPLIER on sup_id
          with SUPPLY on sup_id where quantity > 130;
```

As we have already explained, in RQL users can combine several projects and select clauses together to form a query. We call the combination of any selection or projection applied to a relation file a *retrieval process*. The retrieval process could either be a *nested* retrieval (nested query, see Table 4 again) or a *qualified* retrieval, in which case the connective In connects two subclauses together. Queries can be nested in any depth in RQL. In any case the product of a retrieval process is again a new relation represented as a flat file, which can be further manipulated, assigned to, or displayed as any other file by using the normal Requiem facilities.

In RQL, *subqueries* represent sets of values searched by the existential qualifier connective In. A subquery is a Project expression nested inside an Expose expression. Note, that the subquery expression must represent a single attribute relation, which explains the fact why such subqueries consist solely of Project expressions. The system evaluates the overall query by evaluating the subquery first. In the recent version of RQL, subqueries can have only one level of nesting, which implies that a subquery cannot be nested inside another.

To put the whole thing into perspective, let us consider the query "Find all supplier names and locations of those suppliers who supply more than 150 items of any product." We can construct this query stepwise. We begin by finding all Sup_ids of suppliers that supply more than 150 items of any product. Consequently, we formulate the subquery:

```
project SUPPLY
    over sup_id where quantity > 150;
```

We then find all the names and locations of those suppliers selected by the previous subquery. We achieve this by embedding the above subquery in an outer Expose. Thus, the resulting query is:

```
expose SUPPLIER over
  sup_name, location
        when sup_id is in project SUPPLY
                    over sup_id where quantity > 150;
```

The In connective (or existential predicate) merely stands for the existential qualifier of the relational calculus; it tests for set membership. Here, *set membership* is the collection of values produced by the subquery clause. Note that in our example no enclosing parentheses are required to surround the subquery. By default, the Where predicate associated with an Expose statement is considered to be part of the defined subquery.

In RQL, the In connective also searches a set of values included in a so-called *predicate list*. The predicate list must contain at least two elements separated by commas. Consider the following example of an In predicate list:

```
project SUPPLIER over sup_name, location
            where sup_id in (604, 605, 720);
```

The Where clause in this simple example evaluates to *true* only if the attribute Sup_id attains one of the values contained in the predicate list. Actually, users can formulate the previous Expose query by means of a predicate list when they have an idea of what kind of output they are expecting. RQL defines the In connective to be semantically equivalent to a logical expression involving only disjunctions. Contrary to the In predicate is the Not in connective used in RQL to test for the absence of set membership.

To find all the products that may not be momentarily supplied by any supplier, we use the construct:

```
difference (project PRODUCT over prod_no),
          (project SUPPLY over prod_no);
```

Similarly, to find all products that are currently supplied (assuming that momentarily some products exist that are not yet supplied by any supplier), we use the following query:

```
intersect (project PRODUCT over prod_no),
         (project SUPPLY over prod_no);
```

To achieve the formulation of complex queries, RQL provides the notion of *temporary relation variables*. The output resulting from a given query could be assigned to a specific relation variable through the statement:

assign *<relation – variable>* **to** *<query>*

For example, consider the query "Find all the Sup_ids, products, and quantities corresponding to all the suppliers in the database." This rather complex query could be formulated in two steps. First, users locate all the suppliers, the products that they supply, and the quantities that they have in stock:

```
assign TEMP1 to
  join SUPPLIER on sup_id
      with SUPPLY on sup_id;
```

Then users insert the temporary (*transient*) relation variable Temp1 as part of the final query, which looks like:

```
project TEMP1 over
    sup_id, prod_no, quantity;
```

These kinds of constructs, while they give RQL queries a procedural look, add a lot to the versatility and expressive power of the language. Actually, there are many examples in relational database programming, in which users are forced to create an intermediate transient relation to simplify or even optimize some larger computation. On the other hand, the above query could have been written as:

```
project (join SUPPLIER on sup_id
     with SUPPLY on sup_id)
       over sup_id, prod_no, quantity;
```

The last two examples show that it is possible to formulate the same query in different ways, which is appealing and quite beneficial to users.

In addition to the above, RQL offers users the capability to partition a base table into *groups* such that each group includes all the attributes containing the same value for the Group by attribute. RQL provides a set of special built-in functions to be used in conjunction with the Group by statements. Such built-in functions include the constructs: average (Avg), minimum (Min), maximum (Max), total (Sum), and count (Count). Functions used in conjunction with Group by statements are called *aggregate* functions because they operate on tuple aggregates. Consequently, the Group by statement splits a relation into groups so that any built-in function can by applied to each of these groups. Thus, to find the total quantity of products supplied by each supplier, users write:

```
project SUPPLY
      over sup_id, quantity
           group by sup_id sum(quantity);
```

Table 6 depicts the result of this unconditional group query.

Sometimes it is useful to define a predicate that applies to groups rather than to tuples. The Where clause of RQL used in conjunction with Group by statements facilitates the formulation of such queries. For example, to find the suppliers supplying a total of more than 150 items users write:

**Table 6.
Result of the unconditional Group-by statement.**

Sup_id	Total
605	250
612	200
827	120
855	135

```
project SUPPLY
        over sup_id, quantity
                group by sup_id where sum(quantity) >150;
```

Predicates in the Where clause are internally applied after the formation of groups, so that the aggregate functions can be applied in such clauses. Table 7 illustrates the result of this conditional group query.

Finally, the Display statement of RQL displays on screen a set of adjacent tuples as they are physically stored in the relation data file. Furthermore, the Display statement can be used to position the cursor of a table. Initially, the cursor identifies a position in the table, namely the position just before the first tuple in the relation data file. For example, the statement

```
display +0, 3 using SUPPLIER;
```

displays the first three tuples from the relation supplier and positions the cursor exactly after the third tuple. Displacements used in conjunction with a Display statement could also be negative, which means that the cursor must move backwards. To reset the cursor to its initial position at the beginning of the file, users should use the statement:

```
display using SUPPLIER;
```

Data modification and deletion. At a given update session users specify which relation and which attributes are to be the object of the update operation for that session. Using a predicate selection expression, exactly as in the process of retrieval, users specify the tuples to be updated. If a given selection predicate identifies a subset of tuples in the flat file, users can step through the specified tuples, changing values in one tuple at a time.

Requiem displays on screen the attribute name and value of every attribute in each selected tuple. Subsequently, it prompts users with the corresponding attribute names to insert a new value. When users enter a value for a specified attribute, Requiem checks it for conformance to its definition, by including any required validation checks. Upon entering values for all of the specified attributes, users can scan over the screen for one last visual check before instructing Requiem to accept and store permanently the newly

**Table 7.
Result of the conditional Group-by statement.**

Sup_id	Total
605	250
612	200

specified values. For example, to update the values of the attributes Sup_name and Location for all suppliers currently residing in Zurich, users write:

```
update sup_name, location
        from SUPPLIER where location = "Zurich";
```

The system then prompts users to decide whether they wish to make the changes (temporarily stored in a buffer) permanent. All database updates that are committed by means of a special Commit statement are written on the disk and cannot be undone. The Commit statement is obviously in a position to discriminate between delete and update operations. If users decide not to commit the changes, Requiem undoes all the changes/deletions made during an update/delete session and recovers the original contents of the tuple affected by such an operation. After the updating/deletion of a tuple, users can exercise the option to update/delete more tuples of the same relation or escape.

Deletion follows the same pattern as does updating with one difference: users delete specified sets of tuples. This means that users should delete an entire tuple at a time, and not just a part of a specified tuple. For example, to delete all suppliers currently residing in Zurich, users write:

```
delete SUPPLIER where location = "London";
```

At the beginning of the previous section we mentioned that Requiem imposes referential constraints by ensuring that every foreign key value matches a value of its corresponding primary key. What this statement implies is that the updates or deletions of primary keys may have cascading effects. Consider the consequences of the previous Delete statement, which at first glance seems to be a harmless statement. As usual we assume that prior to the execution of this statement the database looks like that in Table 1. After processing the Delete statement, the database would look like that in Table 8. This difference in appearance implies that the deletion of the primary key SUP_ID = 605, associated with the London supplier, has cascaded to its matching values in the composite relation supply. As a result all supply tuples related to the removed London supplier disappear without any user intervention. Such referential constraints concerning primary and foreign keys also affect Update statements.

Table 8.
The suppliers-and-products database after deletion of the London supplier tuple.

Supplier

Sup_id	Sup_name	Location
612	Schmidt	Zurich
827	Dupont	Paris
855	Dick	Bonn

Product

Prod_no	Prod_name	Weight	Finish
10	Desk	150	Pine
12	Bookcase	100	Cherry
21	Cabinet	80	Oak
25	Sofa	20	Oak
33	Dresser	90	Pine

Supply

Sup_id	Prod_no	Quantity
612	10	200
827	12	120
855	25	135

Table 9.
Use of format files in RQL.

```
Command file = format_file

Name: <Sup_name>
Location: <location>

print using format_file
      Sup_name, location from SUPPLIER;

print using format_file
      Sup_name, location from SUPPLIER into out_file;
```

Table 10.
Use of aliases in RQL.

```
Command file = format_file

Name:     <sn>
Location: <lc>

print using format_file
      Sup_name sn, location lc from SUPPLIER;
```

File manipulation functions. Requiem reads and processes series of valid RQL statements from a *command file*. This file usually contains view definitions, but it could also contain any other retrieval statement. Users can process the statements included in the command file in two ways:

• The name of the file could be passed as a command-line argument (in the Unix sense) before users enter Requiem.

• Alternatively, the command file could also be processed from within RQL by typing a "@" character before the command file name.

Requiem also can process another category of external files. These files, called *format files*, specify a format template into which attribute values should be substituted during a display or print operation. The user's current directory should specify a format definition file and should always have the suffix ".form" following its name. The format template has the form:

Text: <*attribute – reference*>

This means that only two types of information can be included in a format definition file, plain text and attribute references. Text takes any character format provided that it is not enclosed in angle brackets. Users indicate attribute references by placing the name of the desired attribute between a pair of angle brackets. Consider the example depicted in Table 9. Note that the file format_file specifies the format that the attributes should have when displayed on the screen, or when forwarded to an output file called out_file.

In addition to the above, when an attribute name is specified in a Print statement, users can provide an alternate name, called *alias*, for that attribute. This option is useful in cases where it is desired that the relation headers in an output table contain alternate names for the actual attribute names. Alternate attribute names can be met in references to that attribute in the predicate selection clause as well in a format definition file. The syntax for specifying aliases is <attribute – name><alias>. Consider the example depicted in Table 10.

RQL provides two additional file manipulation statements, Import and Export. The Import statement imports tuples from a file into a relation. Import takes the form:

import <*file – name*> [**into** <*relation – name*>];

where <file – name> is the name of the input file internally qualified by ".dat." This input file must contain values of the tuple attributes, with each attribute on a separate line. Subsequently, the tuples are appended to the named relation.

IEEE MICRO

April 1989 issue (card void after October 1989)

Name ______________________________

Title ______________________________

Company ______________________________

Address ______________________________

City ____________________ State______ ZIP ________

Country ______________ Phone (_____) ____________

Please send (Circle those you want)

201 Publications catalog
202 Membership information
203 Student membership information
204 Senior membership information
205 *IEEE Micro* subscription information

Reader Interest
(Add comments on the back)

Readers,
Indicate your interest in articles and departments **by circling the appropriate number** (shown on the last page of articles and departments):

150	159	168	177	186
151	160	169	178	187
152	161	170	179	188
153	162	171	180	189
154	163	172	181	190
155	164	173	182	191
156	165	174	183	
157	166	175	184	
158	167	176	185	

Product Information **1**
(Circle the numbers to receive product information)

1	21	41	61	81	101	121	141
2	22	42	62	82	102	122	142
3	23	43	63	83	103	123	143
4	24	44	64	84	104	124	144
5	25	45	65	85	105	125	145
6	26	46	66	86	106	126	146
7	27	47	67	87	107	127	147
8	28	48	68	88	108	128	148
9	29	49	69	89	109	129	149
10	30	50	70	90	110	130	—
11	31	51	71	91	111	131	—
12	32	52	72	92	112	132	192
13	33	53	73	93	113	133	193
14	34	54	74	94	114	134	194
15	35	55	75	95	115	135	195
16	36	56	76	96	116	136	196
17	37	57	77	97	117	137	197
18	38	58	78	98	118	138	198
19	39	59	79	99	119	139	199
20	40	60	80	100	120	140	200

IEEE MICRO

April 1989 issue (card void after October 1989)

Name ______________________________

Title ______________________________

Company ______________________________

Address ______________________________

City ____________________ State______ ZIP ________

Country ______________ Phone (_____) ____________

Please send (Circle those you want)

201 Publications catalog
202 Membership information
203 Student membership information
204 Senior membership information
205 *IEEE Micro* subscription information

Reader Interest
(Add comments on the back)

Readers,
Indicate your interest in articles and departments **by circling the appropriate number** (shown on the last page of articles and departments):

150	159	168	177	186
151	160	169	178	187
152	161	170	179	188
153	162	171	180	189
154	163	172	181	190
155	164	173	182	191
156	165	174	183	
157	166	175	184	
158	167	176	185	

Product Information **2**
(Circle the numbers to receive product information)

1	21	41	61	81	101	121	141
2	22	42	62	82	102	122	142
3	23	43	63	83	103	123	143
4	24	44	64	84	104	124	144
5	25	45	65	85	105	125	145
6	26	46	66	86	106	126	146
7	27	47	67	87	107	127	147
8	28	48	68	88	108	128	148
9	29	49	69	89	109	129	149
10	30	50	70	90	110	130	—
11	31	51	71	91	111	131	—
12	32	52	72	92	112	132	192
13	33	53	73	93	113	133	193
14	34	54	74	94	114	134	194
15	35	55	75	95	115	135	195
16	36	56	76	96	116	136	196
17	37	57	77	97	117	137	197
18	38	58	78	98	118	138	198
19	39	59	79	99	119	139	199
20	40	60	80	100	120	140	200

SUBSCRIBE TO IEEE MICRO

☐**YES**, sign me up!
If you are a member of the Computer Society or any other IEEE society, pay the member rate of only **$18** for a year's subscription (six issues).

Society: ____________________ IEEE membership no.: ______________

Society members: Subscriptions are annualized. For orders submitted March through August, pay half the full-year rate ($9) for three bimonthly issues.

☐**YES**, sign me up!
If you are a member of ACM, ACS, BCS, IEE (UK), IEEE (but not a member of an IEEE society), IECEJ, IPSJ, NSPE, SCS, or other qualified professional technical society, pay the sister-society rate of only **$33** for a year's subscription (six issues).

Organization: ____________________ Membership no.: ______________

____
Full Signature *Date*

____
Name

____
Street

____
City

____
State/Country *ZIP/Postal Code*

☐ Payment enclosed

☐ Charge to ☐ Visa ☐ MasterCard ☐ American Express

Charge-card number

— — — — — — — — — — — — — — — —

Expiration date

— — — —
Month *Year*

Prices valid through 12/31/89
4/89 Micro

Charge orders also taken by phone:
(714) 821-8380 8 a.m. to 5 p.m. Pacific time
Circulation Dept.
10662 Los Vaqueros Cir.
Los Alamitos, CA 90720-2578

Editorial comments

I liked: ______________________

____

____

____

____

I disliked: ______________________

____

____

I would like to see: ______________________

____

____

____

For reader-service inquiries, see other side.

PLACE POSTAGE HERE

PO Box is for reader-service cards only.

IEEE Micro
Reader Service Inquiries
PO Box 16508
North Hollywood, CA 91615-6508

Editorial comments

I liked: ______________________

____

____

____

____

I disliked: ______________________

____

____

I would like to see: ______________________

____

____

____

For reader-service inquiries, see other side.

PLACE POSTAGE HERE

PO Box is for reader-service cards only.

IEEE Micro
Reader Service Inquiries
PO Box 16508
North Hollywood, CA 91615-6508
USA

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 38 LOS ALAMITOS, CA

POSTAGE WILL BE PAID BY ADDRESSEE

IEEE COMPUTER SOCIETY
Circulation Dept.
10662 Los Vaqueros Cir.
Los Alamitos, CA 90720-9804
USA

The Export statement combines import and export tuples from a relation into a file. If the file name is omitted, output is forwarded to the terminal. Export takes the form:

export <*relation – name*> [**into** <*file – name*>];

View functions. In Requiem views are considered to be dynamic windows in the database structure that allow users to visualize the database differently. A base relation and a view primarily differ in that the base relation physically exists as a relation. It exists in the sense that for each tuple of the base relation there exists a physical counterpart in the physical storage and possibly in physical access paths such as indexes, which directly support this specific relation. By contrast, a view is, as previously stated, a virtual relation. That is, a relation that does not really exist in physical storage but is rather derived in terms of one or more base relations.

Views must not be confused with the term *external view*. Broadly speaking, the external view in Requiem means a collection of several relations, some of which comprise views and some of which are base relations. The external schema consists of combined named definitions of these views and relations; it is finally mapped onto the conceptual schema. A mapping of the external/conceptual schema defines the correspondence between the external or logical records of the external schema and the tuples of the conceptual schema. An external record is, as you may have guessed, not necessarily the same as a physical record.

Let us now examine views in the RQL sense in some detail. In RQL, the statement Define view allows users to define a view. Its general format is as follows:

define view <view_identifier> **as** <qualifying_query>.

The qualifying query may begin with a Select or a Project statement, or it may encompass all the query options described in the earlier box. Actually, when a view is defined, its definition in terms of its base relation is stored in an appropriate system catalog called SYSVIEWS. It is remembered only under its specified view name. Consider the following view definition in RQL:

```
define view London_Suppliers as
  select all from supplier where location = "London";
```

This view is defined in terms of the base relation supplier as depicted in Table 11. When the view LONDON_SUPPLIERS is defined, the select expression is not evaluated at the time of the definition. The view is simply expanded whenever its name is encountered in a legal RQL statement. The view LONDON_SUPPLIERS can then be expanded exactly as indicated in Table 12. Except for the fact that it is impossible to define access paths, the above view behaves as if it were a true relation in all other respects.

Table 11.
An instance of the relation supplier.

Sup_id	Sup_name	Location
611	Jones	London
615	Henry	Paris
618	Rogers	New York
625	Smith	Dublin
633	Bates	London
638	Longfellow	London

Table 12.
The view LONDON_SUPPLIERS.

Sup_id	Sup_name	Location
611	Jones	London
633	Bates	London
638	Longfellow	London

Table 13.
Result of an RQL view operation.

Sup_id	Sup_name
611	Jones
633	Bates
638	Longfellow

In particular, users can interrogate the view or even update and delete attributes or tuples from it if this is permitted. Consider for example the statement:

```
project London_Suppliers over sup_id, sup_name;
```

Table 13 shows the result.

For performance and consistency reasons, such queries on a view are translated into queries on the base relation rather than being executed on a materialization of the view itself.[9] Requiem merges the Project statement, issued by users, with the Select statement that was saved in the SYSVIEW catalog during the process of view definition. From this catalog the system recognizes that the Over statement actually refers to the supplier relation. Hence it executes the Select statement of the view definition first. Subsequently, the result of this query is used as an input relation in the Project query statement. What actually happens is that the result of the view is first stored into a snapshot file and is then used as an input argument by the Project statement in the place of the view identifier LONDON_SUPPLIERS. In the previous example the Project statement behaves just like any normal Project statement in a nested RQL query. As a matter of fact the equivalent operation is:

```
project (select all from supplier where location =
        "London") over sup_name;
```

To users, however, it appears as if a relation actually existed in the database called LONDON_SUPPLIERS, with the attributes and tuples as illustrated in Table 12. Actually, in RQL this table is the relation that is furnished after users call the previously defined view by its name, followed by the query terminator symbol:

```
London_Suppliers;
```

The dynamic nature of views characterizes them. A view that results in a retrieval operation will pass on the data it has received or will store the resulting file as a temporary file not belonging to the actual database. In general, views provide subsets of databases that exist only after a view is executed. For example, LONDON_SUPPLIERS is in effect a "dynamic image" allowing users to visualize the underlying base relation differently. Any changes to the base relation supplier becomes automatically visible at the view level. Furthermore, any permitted changes to the LONDON_SUPPLIERS view will automatically be registered in the base relation. As is implied, users can also perform update or delete operations in addition to search operations.

In general, some drawbacks exist as far as view manipulation is concerned. These drawbacks stem from the fact that views comprise dynamic images of a part of the database. It is not always possible to carry out updates on the contents of a view. To achieve this, users must make sure that this update propagates to the base relations. In case of one-to-one relationships between the tuples of a given view and those of a base relation, updates are feasible. Requiem handles view modifications along the lines of the following rules and of what is being mentioned in Date[6]:

• Exactly one base relation constructs a view. Views created on the basis of other views, as well as views resulting from joins, do not qualify.

• A view represents a simple row and column subset of one underlying table, and any modification such as insertion, deletion, or updating on the contents of the view must result in an equivalent operation on the contents of the qualifying table.

These rules, among others, guarantee that view modifications cannot be performed if a view is the outcome of:

• Join statements,
• Group by statements, or
• aggregate function statements.

Catalog functions. We now turn to the discussion of the system catalogs. As far as Requiem is concerned, a database is composed of both base and derived relations (views). Moreover, each relation being stored in the system can have one or more access paths. It is also sometimes necessary to know which users are authorized to work on which data. Thus, Requiem keeps track of this kind of information by maintaining a set of special tables that constitute the *system* or *database* catalogs.

System catalogs are themselves relations stored in the database, and Requiem allows users to interrogate them and obtain information pertinent to the contents of the database. Catalog manipulation is affected via the use of the system's query language. Any viable RQL statement can operate on the system catalogs. It is however irrational to assume that users could directly modify, insert, or delete the data contained in a catalog. The reason for this rule is quite plausible. Allowing such operations could prove to be extremely dangerous in the sense that it would be far too easy to destroy information inadvertently in the catalogs, so that the system would no longer function properly.

Data catalogs are updated automatically during the phase of data definition or data modification through the Create, Update, Delete, and Purge statements.

The Requiem system catalogs contain descriptions of all the objects in the system. Thus, catalogs contain entries for relations in the system (derived or non-derived), dependency links between different relations in the database, attribute descriptions, and access paths. For example, Requiem contains a catalog table for base relations called SYSTAB. SYSTAB includes an entry, that is, a row, for every table defined in the system, giving among other information:

• RELNAME, the name of the relation,
• CREATOR, the identification of the creator of this relation,
• NATRS, the total number of attributes in that relation, and
• DERIV, an indication that this relation is a view and not a base relation.

For example, Table 14 represents the set of catalog entries resulting from the execution of the following data definition statements:

```
create EMPLOYEE
            (empl_id  (char(5), UNIQUE),
             name     (char(20), SECONDARY),
             address           (char(30)),
             salary            (real(10)));
```

Thus, the definition-creation of a relation inserts a description of the relation being created into the system catalogs. This description comprises multiple tuples in multiple catalog relations. The DBMS requires all of the above information to perform its activities properly and is therefore maintained by the DBMS itself.

Table 14.
The system catalog contents after the creation of the relation employee.

SYSTAB

Relname	Creator	Perms	Deriv	Natrs
Systab	Sys	600	Base	5
Sysatrs	Sys	600	Base	5
Sysview	Sys	600	Base	3
Syskey	Sys	600	Base	4
Employee	Jones	600	Base	4

SYSATRS

Relname	Atrname	Type	Length	Index
Systab	Relname	Char	12	Yes
Systab	Creator	Char	20	Yes
Systab	Perms	Num	5	No
Systab	Deriv	Char	5	No
Systab	Natrs	Num	8	No
Sysatrs	Relname	Char	12	Yes
Sysatrs	Atrname	Char	20	Yes
Sysatrs	Type	Char	5	No
Sysatrs	Length	Num	6	No
Sysatrs	Index	Char	5	No
Sysview	Viewname	Char	12	Yes
Sysview	Base	Char	12	No
Sysview	Text	Char	123	No
Syskey	Fgnkey	Char	12	Yes
Syskey	Relname	Char	12	No
Syskey	Target	Char	12	No
Syskey	Primary	Char	12	No
Employee	Empl_id	Char	5	Yes
Employee	Name	Char	20	Yes
Employee	Address	Char	30	No
Employee	Salary	Real	10	No

Table 15.
Result of a catalog interrogation in RQL.

RELNAME
SUPPLIER
SUPPLY

As far as the retrieval of metadata is concerned the relational directory subsystem provides similar functional capabilities to those of a standard DBMS. Consider the following query:

```
project SYSATTRS
        over relname
              where attrname = "sup_id";
```

with the result seen in Table 15.

Here, the SYSATTRS catalog contains entries for each attribute of every relation in the database (including the catalog tables themselves). RELNAME is the name of the relation, and ATTRNAME denotes the specific attribute required in that relation.

Requiem provides several other system catalogs that provide auxiliary information about user- and system-defined relations. For example, the SYSVIEW catalog contains the description of the virtual relation in terms of existing relations as well as its textual definition, while the SYSKEY catalog provides enough information to associate foreign keys in composite and primary keys in target relations.

Finally, Requiem provides the Extract statement, which gives a tabular list of information pertaining to a certain relation: the attribute names, types, key definitions, size of tuples in bytes. In case users want to check the definition of a relation, they can use the Extract statement and not interrogate the catalogs, as this statement provides concise information that otherwise requires a number of catalog queries.

Authorization and security mechanisms. In Requiem the term *database security* implies the protection of the contents and the logical structure of the database against unauthorized disclosure, accidental modification, or destruction. It is therefore necessary to have a security mechanism to provide protection and thus to permit data to be manipulated only in an authorized manner. To perform any viable RQL operation, users must hold the appropriate access privilege for the operation and operands under question.[6] Examples of access privileges are the retrieval statements Select and Project, the Update and Delete statements, Index statements (that is, to open an index on the base relation concerned), and finally the unrestricted authority Database Administrator, or DBA. Furthermore, Requiem is in a position to utilize authorization mechanisms capable of reconsidering access privileges at any moment, thus according or rejecting access to several parts of the database as required. In RQL the security mechanisms are principally based on the SQL-like Grant and Revoke statements.

At system installation time, a specific user receives DBA privileges, which grant them unrestricted control over the contents and the structure of the database. Any other user creating a relation is automatically granted all applicable privileges on that relation. This is accomplished internally via the Grant statement. Any specific user holding a Grant option can grant that privilege to another group of users, or can revoke an already granted privilege through the Revoke option. Consider the following examples:

```
grant update, delete on SUPPLIER to group DB10;
grant select on PRODUCT to James;
grant DBA to Schmidt, Hoffman;
revoke DBA from Schmidt;
revoke delete on SUPPLIER from group DB10;
```

Authorization mechanisms apply to base relations as well as to views. A special catalog, which can be manipulated by RQL constructs, stores access privileges imposed on relations and views. Each time a Grant, or Revoke, statement is issued, Requiem must first confirm that the user who issued the statement has the right to accord or refuse the privileges specified in the statement. If a user holds the right, the system carries out the intersection of the accorded or refused privileges and the ones their destinator already possesses.

I have presented the internal workings and the functionality of an extensible relational database system called Requiem. This system offers some interesting features not yet available in conventional DBMSs for small- and medium-scale systems. The portable DBMS provides its own tools for query language development, and it can be supported by Unix or non-Unix environments running C. One of the most appealing characteristics of Requiem is its extensibility. The system offers a versatile kernel so users can explore artificial intelligence applications to develop a new breed of easy-to-use interfaces.

Requiem also contains capabilities for further extension and improvement (we consider its present configuration to be minimal); however, we are convinced that Requiem at its present stage can provide a useful facility for personal database systems. Lack of a query editor and a sophisticated user interface are the system's present major drawbacks. We plan to continue to upgrade the entire system implementation to meet these requirements.

References

1. C.J. Date, *An Introduction to Database Systems*, Vol. 1, 4th ed., Addison-Wesley, Reading, Mass., 1986, 574 pp.
2. E. Codd and C.J. Date, "Interactive Support for Non-programmers, the Relational and Network Approaches, *Proc. ACM SIGFIDET Workshop on Data Description, Access and Control*, May 1974.
3. M.P. Papazoglou and W. Valder, *Relational Database Management: A Systems Programming Approach*, Prentice Hall Inc., Englewood Cliffs, N. J., Apr. 1989, 578 pp.
4. B.W. Kernighan and R. Pike, *The UNIX Programming Environment*, Prentice Hall Inc., 1984.
5. M. Stonebraker et al., "The Design and Implementation of Ingres, *Proc. ACM TODS*, Vol., 1, No. 2, 1976, pp. 189-222.
6. C. J. Date, *Selected Writings*, Addison-Wesley, 1986.
7. H.M. Deitel, *An Introduction to Operating Systems*, Addison-Wesley, 1984.
8. D. Tanenbaum, *Operating Systems Design and Implementation*, Prentice Hall Inc., 1987.
9. M.A. Stonebraker, "Implementation of Integrity Constraints and Views by Query Modification," *Proc. ACM-SIGMOD Conf. Management of Data*, San Jose, Calif., May 1975.

Mike P. Papazoglou joined GMD as a senior research scientist in 1984 and currently serves as the leader of Project Intent (Integrated Environment for Distributed Databases). He has initiated and led research projects dealing with the design of multiprocessor systems for the parallel execution of object-oriented languages. He held the position of research associate at the Nuclear Research Center Demokritos in Athens and at the Microprocessor Lab at the University of Dundee. He was a senior lecturer at the University of Patras in Greece. His research interests include parallel processing, distributed operating systems, centralized/distributed databases, semantic and object-oriented data models, and object-oriented programming languages and systems.

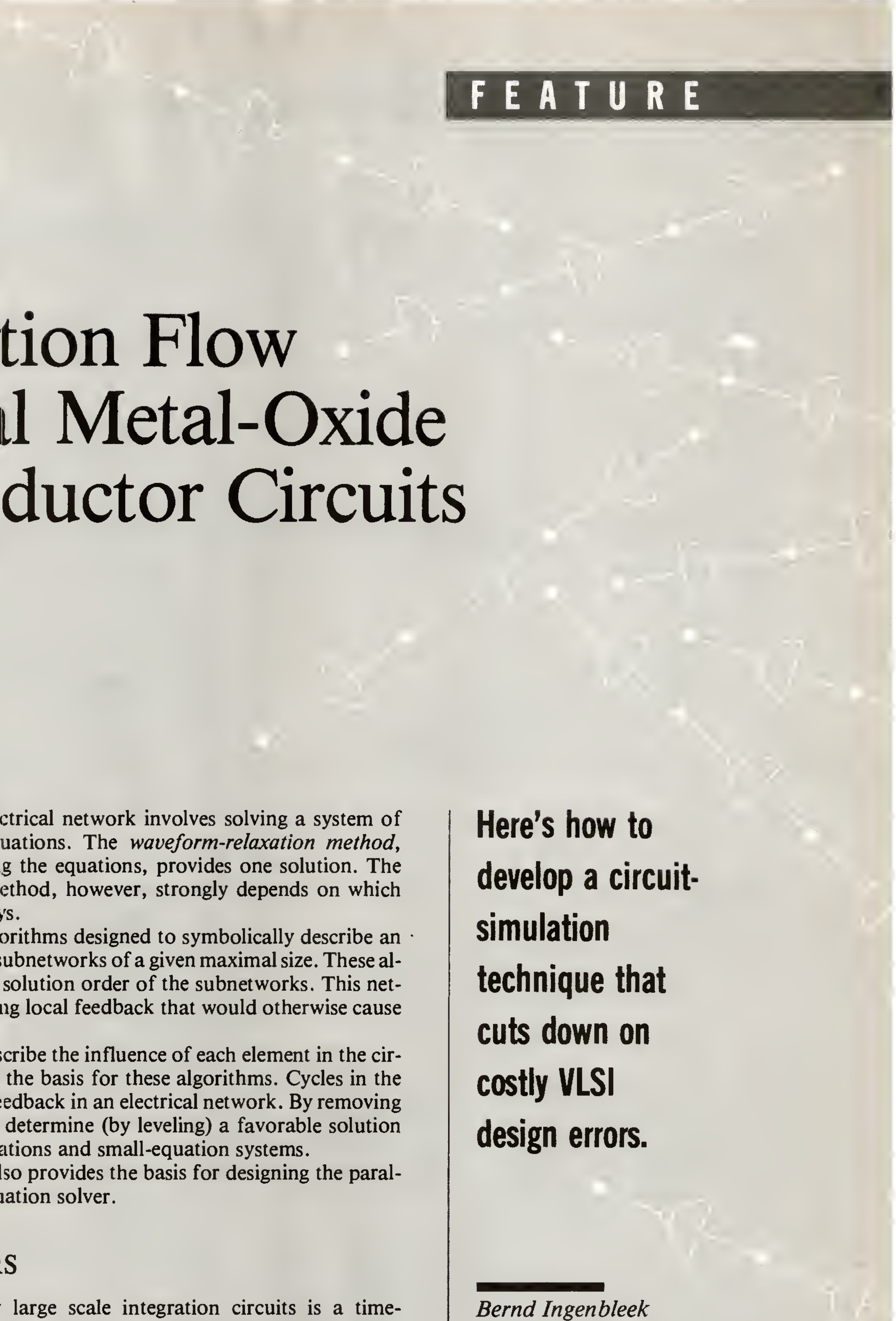
Papazoglou obtained the BSc degree in electronic engineering, the MSc in computer systems engineering, and the PhD in microcomputer systems engineering from the Universities of Dundee and Edinburgh. He has published over 30 technical papers in journals and conferences, authored two books, and served as a referee for scientific magazines and conferences. He is a member of the IEE and the ACM.

Questions regarding this article can be addressed to the author at GMD, F2 Institute fur Systemtechnik, Schloss Birlinghoven, D-5205 St. Augustin 1, West Germany; Email: mike%gmdzi@mcvax.uucp.

Reader Interest Survey

Indicate your interest in this article by circling the appropriate number on the Reader Interest Card.

Low 159 **Medium** 160 **High** 161



Information Flow in Digital Metal-Oxide Semiconductor Circuits

Here's how to develop a circuit-simulation technique that cuts down on costly VLSI design errors.

Bernd Ingenbleek
Anacad GmbH

Klaus Woelcken
GMD (German Research Center for Computer Science)

Claudia Matthaus
Dornier System Gmbtl

The simulation of an electrical network involves solving a system of coupled differential equations. The *waveform-relaxation method,* which entails decoupling the equations, provides one solution. The convergence speed of this method, however, strongly depends on which solution sequence one employs.

Here, we present static algorithms designed to symbolically describe an electrical network and derive subnetworks of a given maximal size. These algorithms further provide the solution order of the subnetworks. This network splitting allows for strong local feedback that would otherwise cause convergence speed problems.

Signal-flow graphs that describe the influence of each element in the circuit on its neighbors serve as the basis for these algorithms. Cycles in the signal-flow graph represent feedback in an electrical network. By removing the cycles, one can naturally determine (by leveling) a favorable solution order for the differential equations and small-equation systems.

The work presented here also provides the basis for designing the parallelization of a differential equation solver.

Circuit equations

The development of very large scale integration circuits is a time-consuming, expensive process. However, avoiding errors in the design phase would provide finished chips in a shorter period of time for less money. Simulation programs based on a description of circuit behavior are important tools for finding these design errors. Error searches carried out at the logical-simulation level give quick, but not entirely accurate, results. More precision occurs in error searches at the physical level. However, the latter takes up considerably more computation time.

To find out what is happening in a powered-up circuit, one must solve n nonlinear, differential equations for $0 \leq t \leq T$ (where n is the number of nodes of the circuit, t denotes time, and T equals the end of simulation).

SPICE[1] does this using the following method:[2]

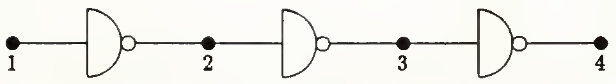


Figure 1. Inverter series.

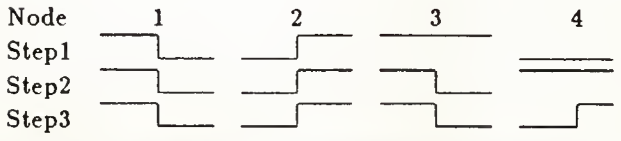


Figure 2. Iteration steps.

```
PROGRAM SPICE
  BEGIN
    read the input;
    find initial values;
    DO FOR $t = 0$ to $T$ STEP $\delta t$
      solve system of equations for time $t$,
      $\delta t$ may vary to force convergence;
    dump any results
  END
```

Lelarasmee et al.[3,4] showed another method for circuit simulation based on the following idea:

```
PROGRAM WAVEFORM RELAXATION:
  BEGIN
    read the input;
    DO UNTIL convergence is reached
      DO FOR $n = 1$ TO number of equations
        solve equation $n$ for $0 \leq t \leq T$;
    dump any results
  END
```

This latter method provides a better convergence than SPICE, but it has two main disadvantages:

- One must solve all other equations before allowing for potential feedback between two nodes.
- A bad sequencing of equations negatively affects convergence, as we further discuss.

Figure 1 shows a circuit that contains three inverters in a series. Imagine an input signal going from High to Low at node 1, with an initial assumption of Low voltages at nodes 2, 3, and 4. Let all equations be solved in a sequence of 4, 3, and 2. Running the waveform-relaxation program results in three steps (see Figure 2).

Step 1. When node 4 is evaluated, it becomes High because node 3 is Low. Node 3 becomes High because node 2 is Low. The node-2 value is the the inverted input signal.

Step 2. Evaluation again starts at node 4, which becomes Low because the result of node 3 is High from the previous step. The node-3 value is the inverted value of node 2. The node-2 value does not change since its input is the same as in the previous step.

Step 3. The nodes are again evaluated in the same manner. Further iteration steps do not change the waveform of any node.

Three waveform iterations are necessary to find the correct result, which a fourth iteration verifies.

If one uses the reverse sequence of equations, the first iteration provides the results, while the second iteration verifies them. The sequence of equations can have a very large influence on computation time, depending on the circuit in question.

The aim of GMD's Simulation of Integrated Circuits (Sisal) project[5] is to develop a fast circuit simulator running on a multiprocessor. We chose a method for solving the equations that is similar to the waveform-relaxation program, but without its disadvantages. The algorithms provide strong local feedback and good sequencing of equations. We also required a circuit-definition language compatible with SPICE and did not allow additional input to specify input/output relations, pass-transistor definitions, or the like.

We developed static algorithms, which means that sequencing does not change during simulation. The algorithms return description groups of locally strong coupled nodes via a single inspection of the circuit. They also return the sequence in which these groups of nodes should be solved. The node grouping can also involve single node groups.

We have had some ideas for dynamic algorithms, which would probably work better because they do take time and data dependencies into consideration. However, these algorithms are the subject of future work in our project. Therefore, this article limits itself to the description of static algorithms based on graph theory.

Basic definitions

Because the developed algorithms are described in terms of graph theory, we present the definitions needed to formulate them, together with some examples. For a more detailed study of graph theory, see Harary[6] and Hein.[7]

Definition 1. A finite, directed graph $G = (V,E,\phi)$ is a triplet of

- a finite, nonempty set of vertices V;
- a finite set of edges E with $V \cap E = \{\ \}$; and
- an incidence mapping $\phi: E \rightarrow V \times V$.

Figure 3a shows a graph with:

$$\begin{aligned} V &= \{v_1,v_2,v_3,v_4\} \\ E &= \{e_1,e_2,e_3,e_4,e_5\} \\ \phi(e_1) &= (v_1,v_3) \\ \phi(e_2) &= (v_3,v_1) \end{aligned}$$

$$\begin{aligned}\phi(e_3) &= (v_2,v_2)\\ \phi(e_4) &= (v_1,v_2)\\ \phi(e_5) &= (v_1,v_2)\end{aligned}$$

Definition 2. If the incidence mapping is injective (there are no multiple edges between two vertices), the graph is called *relational.* These graphs allow a simplified notation because E can be identified with $\phi(E)$: $= \Phi$:

$$G = (v,\Phi), \text{ with } \Phi \subset V \times V, \text{ is a relation.}$$

Definition 3. For better readability, a simplified notation is introduced by:

$$\mathrm{v} \rightarrow \mathrm{w} \overset{def}{\Leftrightarrow} \text{ an edge } e \text{ exists with } \phi(e) = (v,w).$$

In Figure 3b, we have the same graph as in Figure 3a, except that one edge has been removed to obtain a relational graph:

$$\begin{aligned}\mathrm{V} &= \{v_1,v_2,v_3,v_4\}\\ v_1 &\rightarrow v_3\\ v_3 &\rightarrow v_1\\ v_2 &\rightarrow v_2\\ v_1 &\rightarrow v_2\end{aligned}$$

Definition 4. An edge $v \rightarrow v$ is called a *loop.* Both graphs in Figure 3 have a loop at vertex v_2.

Definition 5. A sequence of distinct vertices $v_0, v_1, \ldots, v_n$ with $n \geq 1$ is called an *undirected path* from v_0 to v_n when there are edges $v_{i-1} \rightarrow v_i$ or $v_i \rightarrow v_{i-1}$ for all $i \in \{1, \ldots, n\}$. The length of that path is n.

A sequence of distinct vertices $v_0, v_1, \ldots, v_n$ with $n \geq 1$ is called a *directed path* from v_0 to v_n when there are edges $v_{i-1} \rightarrow v_i$ for all $i \in \{1, \ldots, n\}$. The length of that path is n.

In Figure 3b, one possible undirected path is v_2, v_1, v_3; its length is 2. One possible *directed path* is v_3, v_1, v_2; its length is also 2.

Definition 6. A sequence of distinct vertices $v_0, v_1, \ldots, v_{n-1}$ is called an *undirected cycle* when it is an undirected path and $v_{n-1} \rightarrow v_0$ or $v_0 \rightarrow v_{n-1}$. The length of that cycle is n.

A sequence of distinct vertices $v_0, v_1, \ldots, v_{n-1}$ is called a *directed cycle* (or shorter one cycle) when it is a directed path and $v_{n-1} \rightarrow v_0$. The length of that cycle is also n.

Refer again to Figure 3b. The only possible cycles are v_1, v_3 and v_3, v_1. Both have a length of 2.

Definition 7. When an undirected path connects every two distinct vertices v_i and v_j, the graph is said to be *connected.*

The graphs in Figure 3 are not connected because both have a so-called *isolated vertex* v_4.

Definition 8. A predecessor (or a successor) of vertex v is any vertex that has an edge pointing to (or from) v.

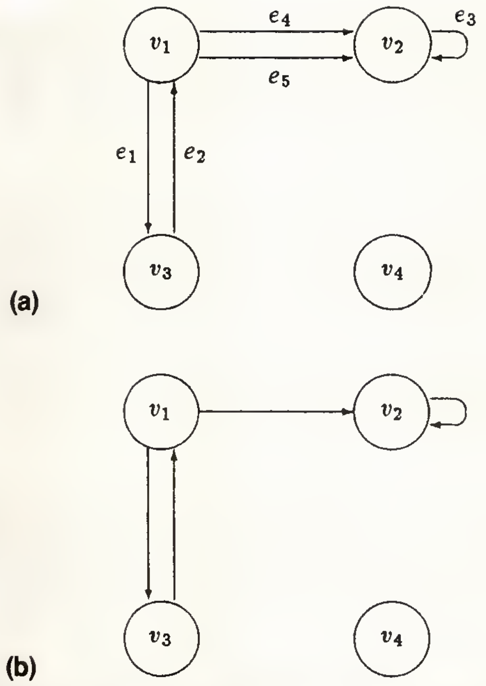


Figure 3. Sample graphs: common (a) and relational (b).

The input (or output) degree of vertex v is defined as the number of edges pointing to (or from) vertex v.

$$\begin{aligned}\mathrm{D}^-(v) &:= \{w|v \rightarrow w\} && \text{is the set of predecessors of } v.\\ D^+(v) &:= \{w|w \rightarrow v\} && \text{is the set of successors of } v.\\ \delta^-(v) &:= |D^-(v)| && \text{is the input degree of } v.\\ \delta^+(v) &:= |D^+(v)| && \text{is the output degree of } v.\end{aligned}$$

Definition 9. A vertex that has an input degree of 0 is called a *source vertex of level 1.* A vertex that has an output degree of 0 is called a *sink vertex of level 1.* A vertex v is called a *source vertex of level* n + *1* when $D^-(v)$ contains only source vertices of levels up to n and at least one predecessor of v is a source vertex of level n. Sink vertices of level $n + 1$ are likewise defined.

Definition 10. A graph with neither source nor sink vertices is called a *closed graph.* As one can easily see, every vertex of a closed graph lies within at least one cycle.

Definition 11. A graph is called *weighted* when a function $w : E \rightarrow \Re$ is defined ($\Re$ equals the set of real numbers). The function value $w(e)$ at some edge is the weight of e.

Graph setup

To describe the operations on an electrical network in terms of graph theory, one must transform it into a graph. It is crucial to discard some of the information embedded in the circuit to obtain a simpler model of its behavior. However, one must avoid oversimplification, which would lead too far away from the original problem. The graph setup consists of two phases: mapping and simplification.

Mapping phase. Using the definitions we have presented, we can set up an information-flow graph from an electric graph. Each electric node becomes a vertex of the information-flow graph. The elements of the circuit create the edges of the information-flow graph as follows.

Each resistor or capacitor between vertices v_i and v_j creates two edges e_k and e_l with

$$\phi(e_k) = (v_i,v_j)$$
$$\phi(e_l) = (v_j,v_i)$$

A weight of 1 is assigned to both e_k and e_l (see Figure 4). The weight of 1 means that the information flow is weak in comparison to that created by transistors. As shown further, most edges generated by transistors have a weight of 2.

A metal-oxide semiconductor (MOS) transistor connected to v_d, v_g, v_s, and v_b (subscripts d, g, s, and b stand for drain, gate, source, and bulk) creates edges and associated weights W as follows:

$$\begin{array}{ll}
\phi(e_{gd}) = (v_g,v_d) & W(e_{gd}) = 2 \\
\phi(e_{gs}) = (v_g,v_s) & W(e_{gs}) = 2 \\
\phi(e_{ds}) = (v_d,v_s) & W(e_{ds}) = 2 \\
\phi(e_{sd}) = (v_s,v_d) & W(e_{sd}) = 2 \\
\phi(e_{bd}) = (v_b,v_d) & W(e_{bd}) = 1 \\
\phi(e_{bs}) = (v_b,v_s) & W(e_{bs}) = 1
\end{array}$$

Figure 5 illustrates MOS-transistor edges.

Voltage sources that are grounded and independent create no edges, because they are also sources of information flow. Thus, it is not good to have information flow through a voltage source.

One can extend the list of elements that create weighted edges, but the ones shown here sufficiently demonstrate the principle.

A finite, nonempty, and generally nonrelational graph of information flow with loops and cycles results when

- the circuit connects topologically, and
- directives exist to create edges from or to each neighboring node for each element type used.

Further, the graph is connected, except for those vertices that correspond to source nodes that are grounded and independent.

Simplification phase. Because the setup graph model is too complex, we simplified it by taking a number of steps.

Step 1. Since nodes connected to grounded, independent sources are known, they need not be computed. Thus, one can delete the associated vertices and their incident edges. The same applies to the ground node. A connected graph results.

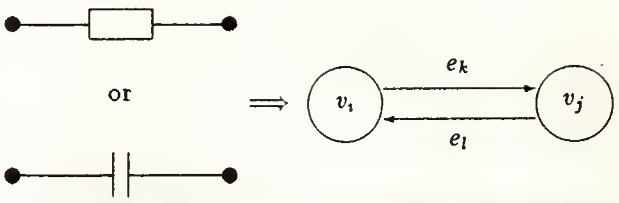


Figure 4. Edges from resistors and capacitors.

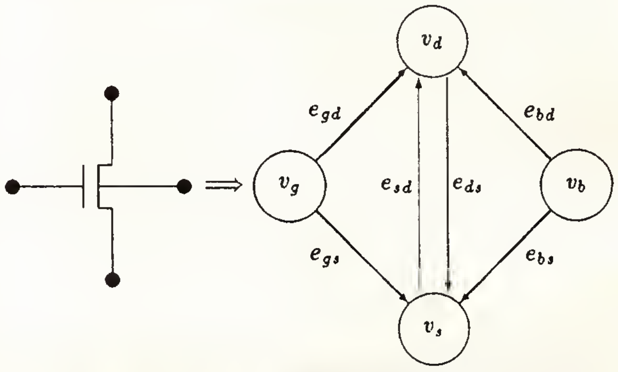


Figure 5. Edges from MOS transistors.

Step 2. A loop at vertex v implies that node v has to be evaluated simultaneously with itself, as is always the case. Thus, all loops are deleted. The result is a graph free of loops.

Step 3. The weights of edges give some measure of the information flow. If multiple edges exist from vertex v_i to vertex v_j, we replace them by a single edge. The weight of this edge is the maximum of the weights of the edges being replaced. We thus obtain a relational graph.

Step 4. For each pair of vertices v_i and v_j with

$$v_i \rightarrow v_j \qquad v_j \rightarrow v_i$$
$$W(v_i \rightarrow v_j) \neq (v_j \rightarrow v_i)$$

the edge with the smaller weight is deleted. The resulting graph takes into account only the strongest information flow between two neighboring vertices. Information that flows in opposite directions between two vertices remains in the graph when the vertices have the same weight.

Step 5. Since the weights are no longer needed, we ignore them.

When these steps have been completed, the resulting graph is nonempty, finite, connected, and relational.

```
program MAP:

begin
    let $\#n$ be the number of nodes;
    let $\#e$ be the number of elements;

    (* generate relational incidence matrix *)
    create square incidence matrix $im$ with $n$ rows;
    initiate $im$ with zeroes;
    do for $e = 1$ to $\#e$
    begin
        if $e$ is a resistor or a capacitor then
        begin
            let $n_1$ be the first node of $e$;
            let $n_2$ be the second node of $e$;
            $im[n_1, n_2] = \max(im[n_1, n_2], 1)$;
            $im[n_2, n_1] = \max(im[n_2, n_1], 1)$
        end;
        if $e$ is a transistor then
        begin
            let $n_d$ be the drain node of $e$;
            let $n_g$ be the gate node of $e$;
            let $n_s$ be the source node of $e$;
            let $n_b$ be the bulk node of $e$;
            $im[n_g, n_d] = \max(im[n_g, n_d], 2)$;
            $im[n_g, n_s] = \max(im[n_g, n_d], 2)$;
            $im[n_d, n_s] = \max(im[n_d, n_s], 2)$;
            $im[n_s, n_d] = \max(im[n_s, n_d], 2)$;
            $im[n_b, n_d] = \max(im[n_b, n_d], 1)$;
            $im[n_b, n_s] = \max(im[n_b, n_s], 1)$
        end
    end

    (* remove independent source nodes and incident
      edges *)
    do for $e = 1$ to $\#e$
        if $e$ is a independent source then
        begin
            let $n_1$ be the first node of $e$;
            let $n_2$ be the second node of $e$;
            remove rows $n_1$ and $n_2$ from $im$;
            remove columns $n_1$ and $n_2$ from $im$;
            $\#n = \#n - 1$
        end;

    (* remove the loops *)
    do for $n = 1$ to $\#n$     $im[n, n] = 0$;

    (* handle antiparallel edges with different weights *)
    do for $row = 1$ to $\#n$
    do for $column = row + 1$ to $\#n$
        if $im[row, column] > im[column, row]$ then
            $im[column, row] = 0$
        else if $im[row, column] > im[column, row]$ then
            $im[row, column] = 0$
        end
    end

end MAP
```

Figure 6. The mapping algorithm.

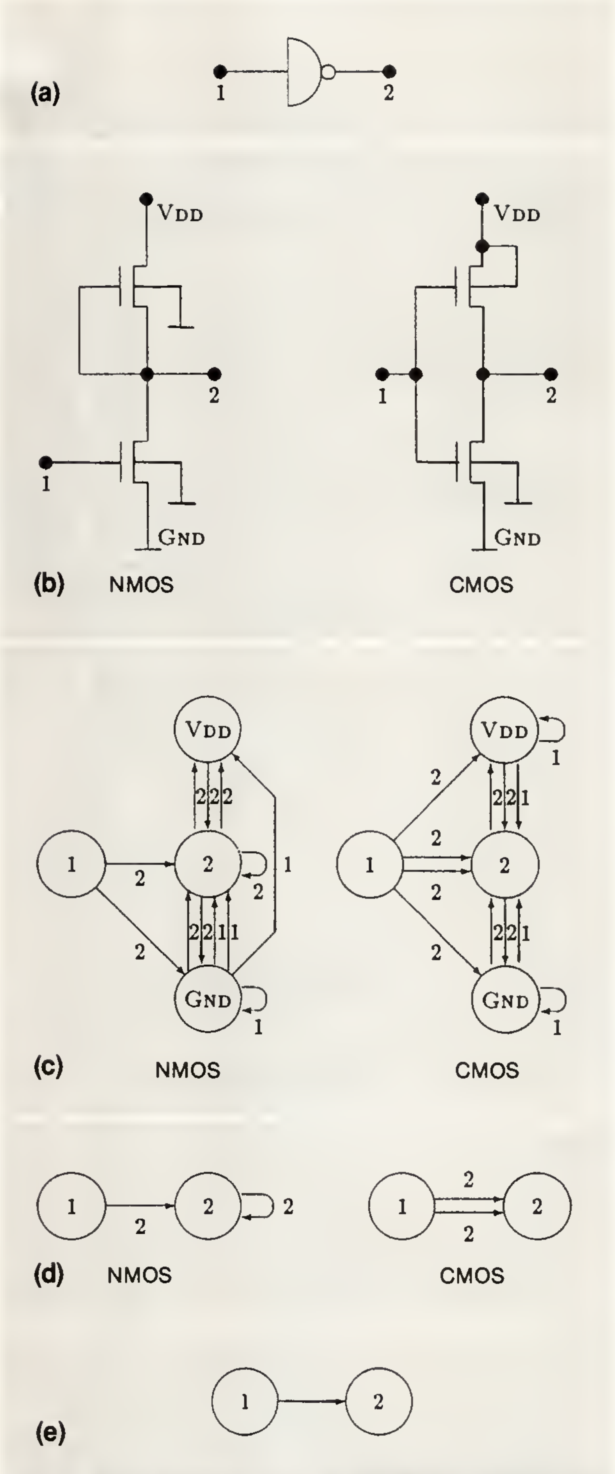


Figure 7. A simple gate and its associated graphs: NOT gate (a), circuit designs (b), circuit-design graphs (c), simplified graphs (d), and further-simplified graphs (e).

The graph generally contains a large number of cycles.
Figure 6 presents the mapping-algorithm program.

Examples of simple gates. Before dealing with a more complex example, let's look at the two simplest gates and their associated graphs.

First note the NOT gate shown in Figure 7a. Figure 7b illustrates its circuit designs in both N-channel MOS (NMOS) and complementary MOS (CMOS) technology. Following graph construction rules pro-

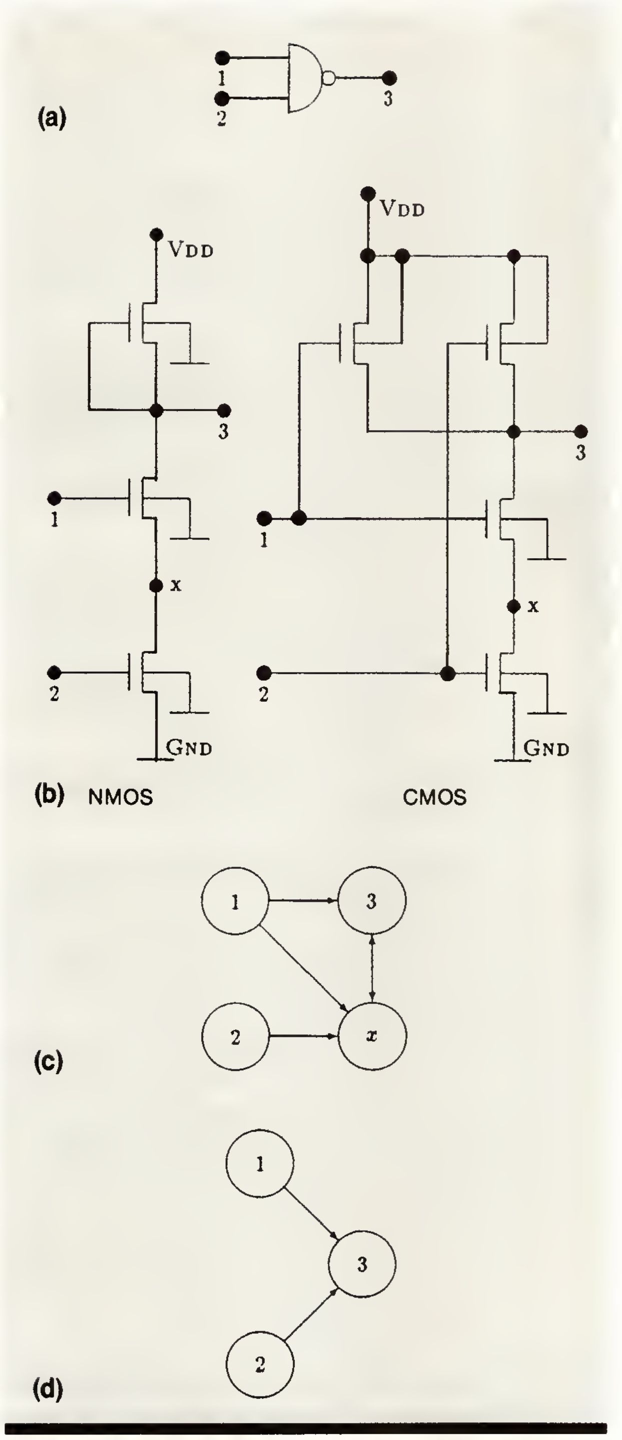


Figure 8. Another simple gate and its graphs: NAND gate (a), circuit designs (b), circuit-design graph (c), and simplified graph (d).

duces the representation shown in Figure 7c. Removing the voltage sources VDD and GND and their incident edges results in Figure 7d. Further simplifying the graphs in both cases leads to the result shown in Figure 7e.

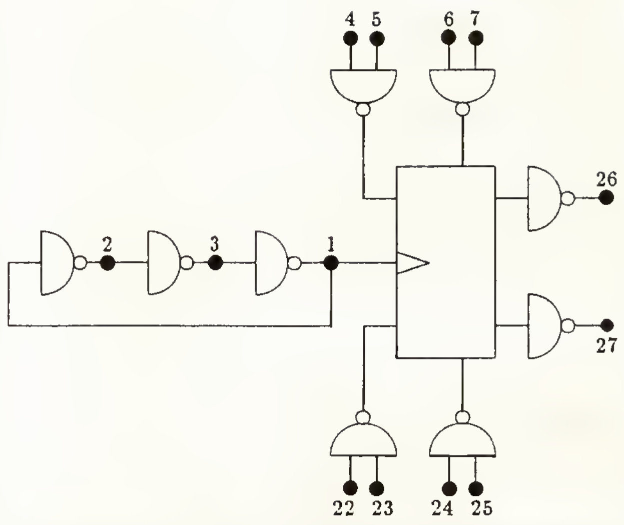


Figure 9. Flip-flop with gates.

This result is just what should be expected from an algorithm that searches the information flow through a NOT gate.

We apply the algorithm to a NAND gate in the same way, as in Figure 8a. Figure 8b shows a NAND gate designed in NMOS or CMOS technology. The resulting graph in both cases appears in Figure 8c. Although the NAND gate is logically symmetrical, note that its technical realization is not.

The next section on grouping shows how to perform graph operations to obtain the form shown in Figure 8d.

Now we are prepared for a more complex example. Suppose a J-K, master-slave, flip-flop (see Figure 9) has an oscillator connected to the clock pin and some gates connected to the other pins. Figure 10 shows this circuit broken down to the gate level. Using what is known from the previous examples, one can quickly derive the information-flow graph. Figure 11 shows the result when NMOS is used.

Grouping

A group is a collection of nodes that are always computed simultaneously during simulation.

Motivation. Solving a system of coupled differential equations that represent an electrical network by the waveform-relaxation method rests on two ideas. First, one must decouple the equations. This means that all node voltages are assumed to be known except the one whose associated differential equation is being evaluated. Every node is evaluated in this manner.

Secondly, the coupling of the equations makes it necessary to solve the system more than once to reach convergence.

In situations where the mutual influence between two nodes is of the same order of magnitude, it is not good to retain the waveform-relaxation method in pure form. It is better to find such local situations by circuit inspection and condense those nodes that have a strong feedback to a single node (here referred to as a *group*). Similar to a normal node—which represents a single differential equation—this group represents a small system of tightly coupled differential equations that can be solved by any integration method for stiff differential equations.

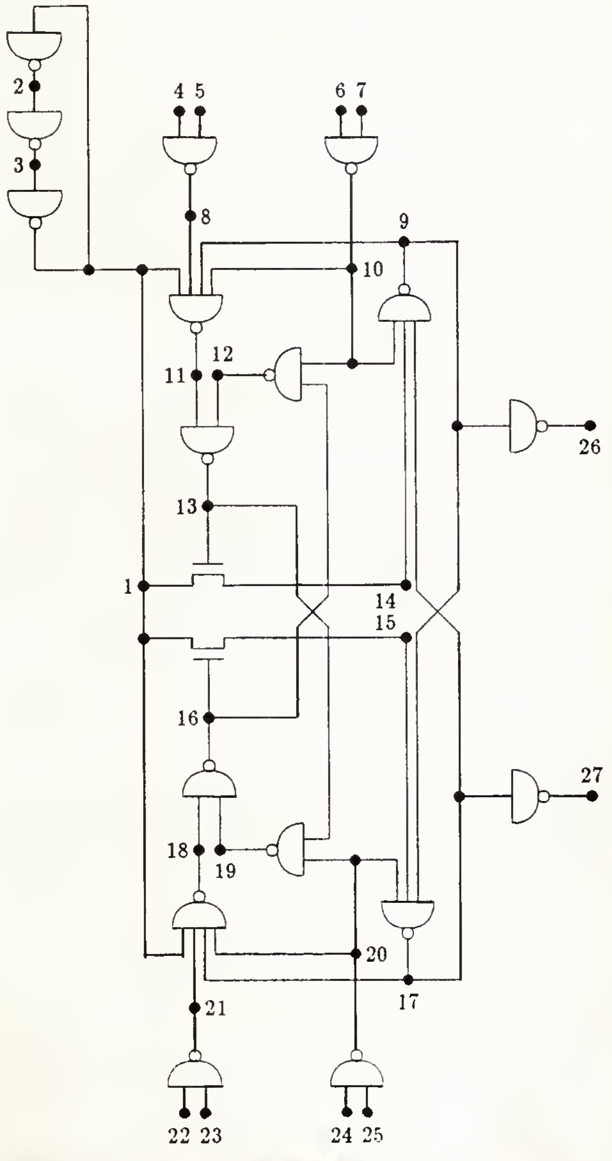


Figure 10. Example circuit (expansion of Figure 9).

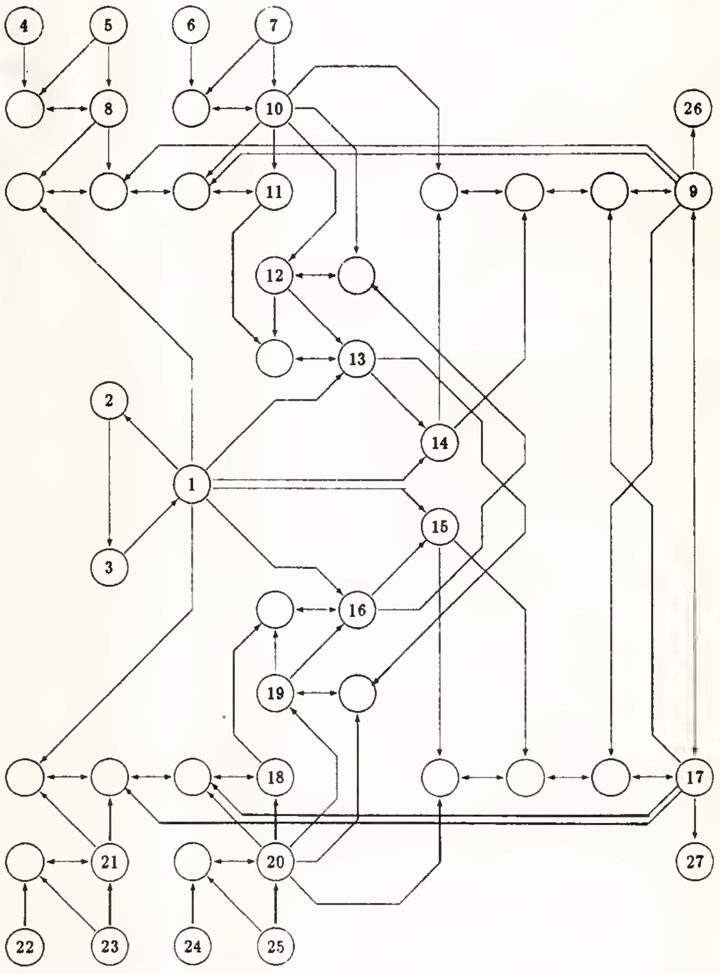


Figure 11. Example graph of the circuit shown in Figure 10.

How can one find such groups? The extreme method would place all nodes of the circuit into a single group. The result is the circuit as seen by SPICE. For faster computation, we introduce a parameter (called *numrec*) to specify the maximum number of nodes that can be placed in a group. From current experiments, we assume that values for numrec of about 4 to 10 give the best results.

Although the grouping algorithm was developed from a problem viewed in terms of graph theory, it usually decomposes—somewhat dependent on numrec—a circuit to logical gates.

Method. In terms of graph-grouping methods, the idea is to find cycles of lengths up to the value of numrec.

Finding cycles in graphs usually takes a lot of computation time, but that is not important here because

```
program GROUP:

begin

    (* initialisation *)
    let $\#n$ be the number of nodes;
    do for $n = 1$ to $\#n$
    begin
        create a group only containing node $n$;
        put that group into the list of groups
    end;

    (* union of groups *)
    let $g$ be the first group of the list of groups;
    while $g$ exists do
    begin
        call UNION($g$);
        let $g$ be the next group of the (evtl. changed)
          list of groups
    end

end GROUP


program UNION($g$):

begin

    (* start with cycles of length 2 *)
    let $l^+$ be the set $D^+(g)$;
    let $l^-$ be the set $D^-(g)$;
    do for $i = 2$ to $numrec$
    begin
        do for $k = 1$ to size of $l^+$
        begin
            let $n^+$ be the $k^{th}$ node in $l^+$;
            let $g^+$ be the group that contains $n^+$;
            do for $j = 1$ to size of $l^-$
            begin
                let $n^-$ be the $j^{th}$ node in $l^-$;
                let $g^-$ be the group that contains $n^-$;
                if $g^+ == g^-$ then
                begin
                    providing that the total number of
                    nodes will not exceed $numrec$ create
                    new group by union of $g$, $g^-$, and
                    all groups between them;
                    return
                end
            end
        end;

        (* prepare search of longer cycles *)
        if $i$ is odd then
            let $l^-$ be the new set $\bigcup D^-(x), x \in D^-(g)$
        else
            let $l^+$ be the new set $\bigcup D^+(x), x \in D^+(g)$;
        memorize the current subgraph
    end

end UNION
```

Figure 12. The group algorithm.

the maximum length of cycles is not very large. Thus, the search can end after a few steps.

To find cycles of length 2 at vertex v, we construct the sets P and S.

$$P := D^-(v) \quad \text{(the predecessors of } v\text{)}$$
$$S := D^+(v) \quad \text{(the successors of } v\text{)}$$

If any length-2 cycle occurs, the set $P \cap S$ is not empty. Each element of $P \cap S$ connects via a cycle of length 2 to vertex v.

Next, the set of all successors of each element of S replaces set S. Now $P \cap S$ contains all nodes that connect to v by a cycle of length 3. To know the third vertex of each cycle, one must keep in mind which element of the previous S created the elements of S.

By alternately replacing P and S with their predecessor or successor sets, one can find cycles of any length within a quadratically increasing computation time. As we are only interested in short cycles (up to lengths of about 10), this algorithm is sufficiently fast for our needs (see Figure 12).

Application. Table 1 shows one possible result of applying the grouping algorithms—with a numrec that equals 4—to the graph shown in Figure 11. Figure 13 demonstrates the result.

Sequencing

As discussed, sequencing is vitally important to the convergence speed of the waveform-relaxation method.

The final step. When one completes graph simplification and grouping, the resulting graph generally contains cycles. Although the graph has become simpler, the initial problem still remains. Inspecting the graph reveals three different types of vertices:

- level-1 source vertices (no edges end here),
- level-1 sink vertices (no edges start here), and
- vertices with input and output degrees greater than zero.

The fourth possible type—with input and output degrees of zero—does not occur because an isolated vertex in a connected graph is impossible. Also, this type of graph consists of only this vertex—it has no edges. One can easily find a good sequence for just one single vertex.

Leveling. The level-1 source vertices are the information sources. One can immediately compute their associated electric nodes because their behavior does not depend on any other vertices in the graph. We deleted these information sources from the graph and put them into a sequence list.

As these source vertices don't influence each other, one can compute them in parallel on different processors.

Table 1.
Results of grouping algorithms.

Group	Vertices
A	1, 2, and 3
B	8 and the one to its left
C	10 and the one to its left
D	11 and the three to its left
E	9 and the three to its left
F	12 and the one to its right
G	13 and the one to its left
H	21 and the one to its left
I	20 and the one to its left
J	18 and the three to its left
K	17 and the three to its left
L	19 and the one to its right
M	16 and the one to its left

Because the successors of each source vertex have lost at least one input edge, they can become new source vertices. In that case, we remove them from the graph and append them to the list. This process reoccurs until no more source vertices remain.

If we start with a graph free of cycles, the indicated iterative process deletes all vertices from the graph, and we find the sequence.

The level-1 sink vertices are the information sinks. One should compute their associated electric nodes last because no other vertex behavior depends on the sinks. Thus—in the same way as the graph has been leveled starting from the sources—the sinks of any level are deleted from the graph. But because the sinks are computed in the opposite order (level-1 sinks last), they are pushed into a stack instead of being placed on a list.

As with the sources, one may evaluate all sink vertices of the same level in parallel on different processors.

Assuming that we have the nontrivial case of a graph with at least one cycle, the graph that remains after the leveling process is a closed graph.

Alexander's method. As we want to apply the waveform-relaxation method to the remaining network, we must find some sequence of computation from the remaining graph. But there is no "natural" way to do this for a closed graph.

Suppose the graph is a single cycle. Then, because of its symmetry, it does not matter where the cycle is broken up to define a linear graph. Any vertex is as good (or bad) as any other for selection as the starting point.

Starting points that are more or less suitable can exist in a more complex graph, but one cannot find them without taking their electrical properties into account.

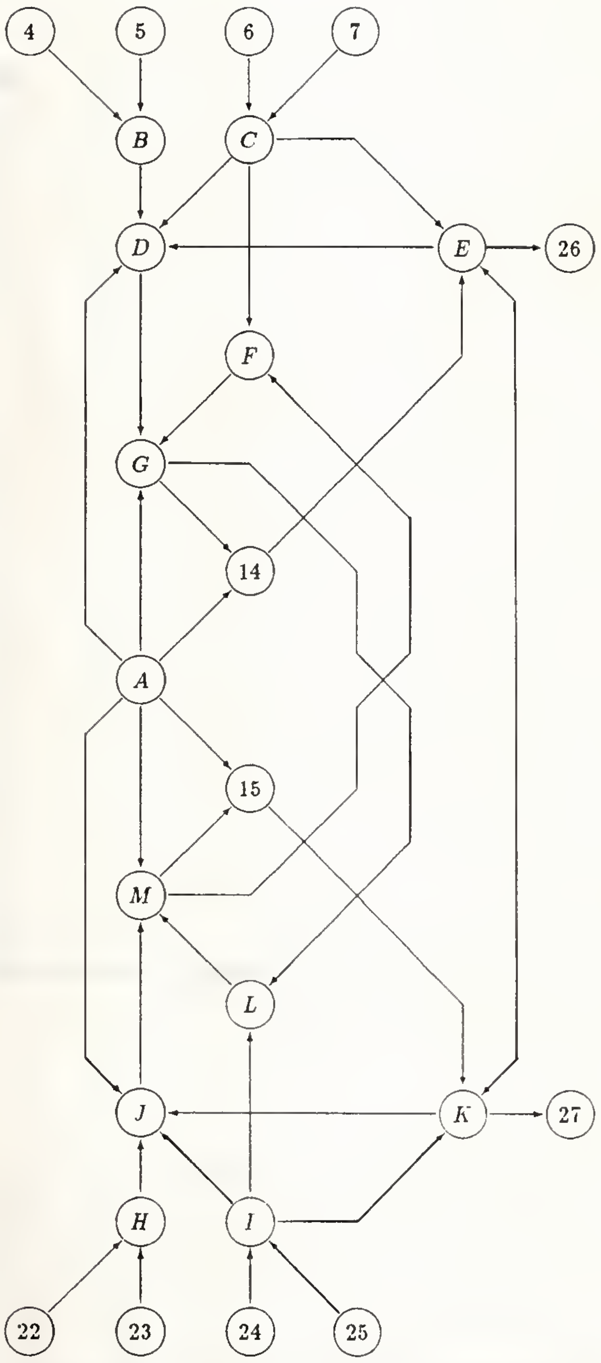


Figure 13. Example of a grouped graph.

The best way would be to simulate the circuit, but that is exactly the problem we started with.

Because of the lack of modern algorithms for cutting cycles, let's try the classical method used by Alexander the Great, who cut the Gordian graph. One selects an arbitrary edge of the graph and cuts through it with a

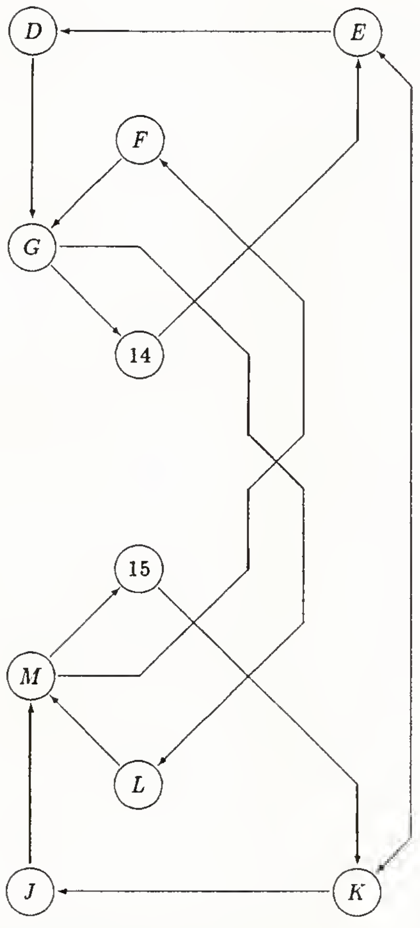


Figure 14. Example of a closed graph.

sword (thus deleting it). The resulting graph is either a still-closed (but less complex) graph, or a graph with new information sources and sinks. Since the sources and sinks are incident to the edge just removed, they are easy to find. In the first case, one applies Alexander's method again to the remaining graph until the second case occurs. Otherwise one uses the leveling process described in the previous section. Repetitive application of these two algorithms makes the graph disappear into a sequence list and stack.

Example of sequencing. Now note the example graph in Figure 13. The source vertices are

- level 1: 4, 5, 6, 7, 22, 23, 24, 25, and *A*; and
- level 2: *B*, *C*, *H*, and *I*.

The sink vertices on level 1 are 26 and 27. The remaining graph is the closed one in Figure 14. All vertices are members of at least one cycle.

Next, edge $E \rightarrow D$ is selected for deletion. This cut creates the source *D*, which is deleted from the graph. One way to continue processing the graph until it disappears is to cut:

- $G \rightarrow 14$, which deletes source 14;
- $E \rightarrow K$, which deletes source *E*;
- $F \rightarrow G$, which deletes sources *G*, *L*, and sink *F*; and
- $15 \rightarrow K$, which deletes sources *K*, *J*, *M*, and 15.

At this time, all vertices reside in a sequence list or stack. Popping up the stack and appending its elements gives the final sequence in which the differential equations and their systems (represented in Figure 13) should be solved. The braces indicate possible parallelization:

- {4, 5, 6, 7, 22, 23, 24, 25, *A*},
- {*B*, *C*, *H*, *I*},
- {*D*},
- {14},
- {*E*},
- {*G*},
- {*L*},
- {*K*},
- {*J*},
- {*M*},
- {15},
- {*F*},
- {27, 26}.

The algorithms we have presented distinguish themselves from other methods through a number of characteristics.

The SPICE format functions as the sole input description. No other circuit specifications such as input-output relations or pass transistors are required. The algorithms are completely technology independent. For example, the present implementation can easily extend to circuits with bipolar components.

The weights used in the signal-flow graphs currently depend only on element type. We used this method to maintain program simplicity. One could derive these weights from the actual circuit components to obtain a more representative signal-flow graph.

The computing time for the algorithms grows linearly with the size of the simulated circuit. This fact is particularly relevant for the simulation of large integrated circuits. Ploeger, Klaassen, and Paap[5] provide the results of our performance measurements.

Our method recognizes circuit paths that are computable in parallel. This approach simplifies the implementation of a simulation program on parallel hardware.

References

1. L.W. Nagel, "Spice2: A Computer Program to Simulate Semiconductor Circuits," Tech. Memo No. ERL-M520, Univ. of California, Berkeley, May 1975.
2. D. Cohen, "Program Reference for Spice2," Tech. Memo No. ERL-M592, Univ. of California, Berkeley, June 1976.
3. E. Lelarasmee, A.E. Ruehli, and A.L. Sangiovanni-Vincentelli, "The Waveform Relaxation Method for the Time-Domain Analysis of Large Scale Integrated Circuits," *IEEE Trans. Computer-Aided Design of ICAS*, Vol. CAD-I, No. 3, Aug. 1982.
4. E. Lelarasmee and A.L. Sangiovanni-Vincentelli, "Relax: A New Circuit Simulator for Large Scale MOS Integrated Circuits," *Proc. Design Automation Conference*, IEEE CS Press, Los Alamitos, Calif. (microfiche), June 1982.
5. P.G. Ploeger, B. Klaassen, and K.L. Paap "Simulating Electrical Circuits Using SISAL," *ASIM 87*, 1987.
6. F. Harary, *Graph Theory*, Addison-Wesley, Reading, Mass., 1969.
7. O. Hein, *Graphentheorie fur Anwender*, B.I.-Wissenschaftsverlag, 1977.

Bibliography

Hachtel, B., "A Survey of Third-Generation Simulation Techniques,"*Proc. IEEE*, Vol. 69, No. 10, Oct. 1981.

Newton, A.R., "Relaxation Based Electrical Simulation," *IEEE Trans. Electron Devices*, Vol. ID-30, No. 9, Sept. 1983.

Newton, A.R., "The Simulation of Large-Scale Integrated Circuits," Tech. Memo No. UCB/ERL-M78/52, Univ. of California, Berkeley, July 1978.

White, J., "Relax2.1: A Waveform Relaxation Based Circuit Simulation Program," *Proc. Custom Integrated Circuit Conference*, May 1984.

Bernd Ingenbleek is now technical director of Anacad GmbH, where he develops CAD systems. He was one of the main researchers in the Sisal simulator system-development project at GMD. His main interests are semantic networks, object-oriented user interfaces, and very large scale integration design.

Ingenbleek holds the Diplom Physik degree from Bonn University, Germany.

Klaus Woelcken joined GMD in 1972 and thereafter became responsible for its computing center in Bonn for IBM mainframe and MVS operating systems. From 1980-83, he led a scientific-advisers staff that reported to the executive board of GMD. Woelcken also developed the conceptual framework for a CAD research group in microelectronics. This group provided the impetus for the national Design of Integrated Circuits program begun in 1983 to support West German universities in the education of designers and CAD specialists. The program also provided access to process lines for integrated-circuit prototyping. His department served as the basis for a 1985 joint venture between GMD and Siemens to make chip design possible for West German universities and research institutes. Since 1987, he has coordinated a similar project for the European Communities.

Woelcken holds a doctors degree in nuclear physics from the University of Giessen, Germany.

Claudia Matthaus worked at GMD as a systems programmer in the Sisal project and on other VLSI topics. She is now a software engineer at Dornier System Gmbtl.

Matthaus received a Diplom degree in computer science (Informatik) from the University of Bonn.

Questions about this article can be directed to Klaus Woelcken, GMD, Postfach 1240, Schloss Birlinghoven, D-5205 Saint Augustin 1, West Germany.

Reader Interest Survey

Indicate your interest in this article by circling the appropriate number on the Reader Interest Card.

Low 162 **Medium** 163 **High** 164

Micro News

Micro News features information of interest to professionals in the microcomputer/microprocessor industry. Send information for inclusion in MicroNews one month before cover date to Managing Editor, IEEE Micro, *10662 Los Vaqueros Circle, Los Alamitos, CA 90720-2578.*

Talking glove speaks for the handicapped

Christine Miller, Assistant Editor

An electrical engineering doctoral student at Stanford University wants to help nonvocal, deaf-blind people communicate "because they can be so isolated." James Kramer is working on a device he calls the *talking glove*, which he hopes will end this isolation through microelectronics. As Kramer puts it, "I wanted to make a device people would *need* rather than just *want*, a device that would enable them to do something they couldn't do before."

Kramer's prototype system works as follows. Deaf users wear an instrumented glove and use their fingers to spell individual letters to put across an idea. The glove holds strain-gage flex sensors that measure and register a change in the resistance of affected glove material when the user bends a finger to spell. The resultant signal is then amplified and sent to a computer. A Motorola 68HC11 microcomputer receives finger-joint-angle information from the glove and sends it to a Zenith laptop for processing by a recognition algorithm. This algorithm selects the most probable letter of the finger-spelling alphabet. A speech synthesizer converts the recognized letter and outputs the speech in words to a hearing person through a speaker pendant worn by the deaf user.

The hearing person replies to the deaf user by typing on a pocket-sized keypad. If the deaf person is sighted, the message displays on a Seiko LCD monitor worn as a wristwatch. Deaf-blind users can retrieve the information on a specially designed mechanical braille display worn at the waist.

At present, all devices are wired to a laptop through an interface box. Kramer plans to design a fully portable system by using a battery-operated Sony Walkman-type of microcomputer (worn by the deaf user) and employing either an infrared or radio link.

James Kramer wears the talking glove, a speaker pendant, and an LCD display. Fellow student Sylvia Plevritis types her message onto a small keypad. (Photo by Louis Lerman.)

Kramer describes the project so far. "Two main technical challenges were hand in glove, so to speak: the design of the actual sensors in the glove and the interpretation of joint-angle information to correspond with a letter of the alphabet. The fingerspelling recognition algorithm of the driving software was also very challenging. I'm still optimizing that algorithm. Observing handicapped people use the devices is helping me refine the system. My sign-language teacher, who is deaf, suggested how to make the glove more comfortable so that it could be worn on a daily basis."

Kramer explained that he has been working on the project for several years in conjunction with Larry Leifer, professor of mechanical engineering. "Part of my original inspiration was a device developed by Stanford engineering students called *Dexter*. Users type into a keyboard and a mechanical hand performs fingerspelling that deaf-blind persons can feel. What I'm working on right now is a complement to that. I realized that Dexter was just half the communication. There still needed to be a way that nonvocal persons could express themselves to hearing persons."

What's next? Future plans include using a 16-bit rather than an 8-bit processor. Kramer adds, "My main goal is to develop a system that recognizes sign-language signs, which are another order-of-magnitude more difficult to recognize than fingerspelling." As probably so many researchers feel at one time or another, Kramer capsulized his project with the thought: "It will reach a stopping point, but there will always be more work to do."

Funds for Kramer's research come from the Fannie and John Hertz Foundation, the Veterans Administration, and Stanford's Center for Design Research.

In a related event, a Redwood City company, VPL Research, Inc., has sued Kramer and two other graduate students for compensatory and punitive damages totaling $500,000. VPL claims Kramer infringed the company's glove-technology property rights by violating a nondisclosure agreement he signed during a visit in either 1986 or 1987.

Kramer has categorically denied the charges and states that the talking glove uses a sensor technology completely different from VPL's. He has assigned the rights for the glove to Stanford University, which has applied for a patent. At this writing, no court decision has taken place.

NEC v. Intel: A new approach to reverse engineering?

Richard H. Stern

The recent trial court opinion in *NEC Corp. v. Intel Corp.* differs significantly from prior US cases in its treatment of reverse engineering of software. Put in a nutshell, the NEC/Intel decision tolerates reverse engineering of software in circumstances in which prior decisions would have thoroughly condemned it. Moreover, the decision considers efforts to produce compatibility a legitimizing motivation. Earlier precedent would have considered that action—at best—to be irrelevant in the determination of copyright infringement.

The NEC/Intel litigation concerns NEC's attempt to reverse engineer the Intel 8086 and 8088 microprocessor chips. NEC wanted to understand the microcode computer programs stored in the chips and develop competitive products that users could substitute in their equipment for Intel chips. Both these aspects of reverse engineering have faced serious obstacles from US copyright law in the past. The NEC/Intel decision, however, finds ways to overcome or ignore these obstacles.

Intel charged NEC with copying the microcode contained in the 8086 and 8088. NEC countered with a barrage of defenses. The court held that the microcode of microprocessor chips is a form of computer program, and therefore is subject to copyright protection. It rejected NEC's claim that microcode was outside the scope of the copyright act. The court then held, however, that Intel's failure to ensure proper copyright marking of the chips resulted in forfeiture of Intel's copyright. In addition, the court held that NEC's microcode—and NEC's acts in relation to the microcode—did not infringe Intel's copyright.

The court recognized that to understand a computer program stored in a piece of hardware (here a chip), one must typically extract the program from the hardware and place it into a form in which it can be read. One cannot read a chip in the same way that one reads a book—by simply looking at it. The computer program is stored in a chip in a way that requires machine and software intervention before the program is readable. Such a reading requires the unloading and listing of the code of the program. In addition, the court recognized, the unloaded program must be disassembled or otherwise translated from a form consisting of 1s and 0s (or hexadecimal code or other equivalent) into a more intelligible form.

NEC unloaded, listed, and disassembled the microcode computer programs from Intel's chips. NEC then prepared successive versions of the programs that were ultimately utilized in their V20 and V30 adaptations of the 8088 and 8086. The initial versions were apparently substantially similar to Intel's computer programs, but the final versions were not.

NEC successfully marketed its V20/V30 in competition with the 8088/8086. The V20/V30 were enhancements of the 8088/8086. However, a customer could replace Intel chips with NEC chips without any adverse effect.

In departure from earlier precedents, such as *Apple v. Franklin*, the NEC/Intel decision appears to find functionality and legitimate reverse engineering in circumstances in which other courts would probably have found unjustified copying and thus infringement. For example, designers had added a microcode "patch" to the 8088 as an ad-hoc bug fix. No similar bug existed in the related 8086—its microcode did not need the patch. When NEC designed its chips to compete with the 8088 and 8086, it placed the bug into the V20 and incorporated a similar patch into its microcode.

Intel urged that this showed "slavish copying," since a chip with a bug was not necessary to accomplish the purposes of the 8088. This copying also resulted in different microcodes for the V20 and V30 (as in the 8088 and 8086 microcodes) instead of the same microcodes, which would normally make more sense for related chips. Under the Apple/Franklin definition of functionality, this feature would surely have been nonfunctional. In that case, the court held: "Franklin may wish to achieve total compatibility with independently developed application programs written for the Apple II, but that is a commercial and competitive objective which does not enter into the . . . issue of whether ideas and expressions have merged." Even if particular lines of code were necessary to achieve compatibility, that fact did not make the code functionally dictated. (Such code would be part of an *unprotected idea*—rather

***IEEE Micro* editor-in-chief Joe Hootman presents James J. Farrell III, retiring EIC, with gifts from *Micro*'s editorial board members on January 26. Farrell received two bound volumes containing the magazines produced during his four years in office, as well as a briefcase.**

than part of a *protected expression.*) Yet, the NEC/Intel court held that NEC was justified in following its design procedure because it wanted to make a V20 that was completely hardware compatible with the 8088, with interchangeable chips.

In addition, the court refused to find any copyright infringement in NEC's listing and disassembling of Intel microcode to understand it and decide how to make a competitive product. NEC designed its ultimate V20/V30 microcodes by an iterative process after disassembling the Intel microcodes. The court acknowledged that the first iteration (version 0) of the NEC microcode was based on the disassembled Intel microcode and was substantially similar to it. However, the version-2 iteration that NEC ultimately used in the V20 microcode was not substantially similar to Intel's, although version 2 was a lineal descendant of version 0.

The court stated that "NEC's final version of the challenged microcode . . . [was] the only one against which a claim of infringement may be directed." Since version 2 was not substantially similar to Intel's microcode, no copyright infringement existed. In effect, the NEC/Intel court refused to find that version 2 was an infringing "derivative work" based on Intel's work. This finding held even though version-2 code was in a sense "derived from" and based on Intel's microcode. The reason: NEC's final microcode did not include protected elements (as in protected expression) found in Intel's code.

In the past, the courts have based copyright infringement on the simple unloading and listing of a computer program (*Hubco Data Products Corp. v. Management Assistance Inc*). Some opinion holds that modification of a program would probably lead to the creation of an infringing derivative work, regardless of whether the later program was substantially similar to the original.

The NEC/Intel decision therefore appears to be a very favorable development for proponents of reverse engineering of software. The decision provides new legal precedent in support of reverse engineering. This precedent supports understanding software and developing a competing or complementary product as legitimate business activities. Moreover, the NEC/Intel decision indicates that commercial desire for compatibility is a motivation that courts should consider as tending to legitimize conduct.

Micro Bits

AT&T Microelectronics and **Intel Corp.** have entered a five-year agreement to sell each other's existing local area network and ISDN chips. The companies also plan to mutually define, develop, and manufacture future-generation LAN and ISDN devices.

MIPS Computer Systems has signed an OEM agreement with Japan-based **Zuken Inc.** to market the MIPS M/120 RISC as a router accelerator for Zuken printed-circuit-board, CAD-workstation products. Zuken markets products in the US, Europe, and Southeast Asia as well as in Japan.

Engineers from Texas A&M University plan to build computerized factory models of **Sematech** (the US chip-making consortium) to predict its behavior under various operating conditions and product mixes. The research is part of a five-year, $1 million-a-year pledge by Sematech to Texas A&M and the University of Texas, Austin, under the Texas Center of Excellence.

General Electronics Ltd. of Hong Kong appears close to announcing a series of new OEM contracts for The Complete PC to bring in revenue of $100 million by 1990. They are expanding manufacturing facilities in China to total 200,000 square feet.

Matsushita Electric Industrial Co. has developed a 64-bit RISC for use in parallel processing applications, and **NEC Corp.** announced a general-purpose 32-bit microprocessor with internal clock speeds of 45 MHz.

The **3Com Corporation** and **Telesystemes Reseaux** are codeveloping an X.400-based gateway and X.25-based router for international electronic mail. Workstations on 3+ and 3+Open networks can exchange mail with IBM's Professional Office System (Profs), DEC's All-in-1, and Telenet's Telemail, among others.

Problems, problems

We may never microwave gourmet dinners inside our laptops—or compact our garbage through a fax (although some people may disagree with this last statement). But a certain amount of micro-unification is in the product stage. And now—on the other end of packing more punch on a chip—researchers are trying to design just one power source for most of our electrical needs.

The resulting data rate of some one billion bits per second from a single transmission facility obviously requires extemely fast switching. One answer could be optical switching, which is inherently faster than electronic switching but currently has some bugs.

When pulses of light are used for transmission, switching a signal to a different branch in the network requires converting faster photons into slower electrons. The speed mismatch between transmission and switching presents a terrible bottleneck.

Scientists at the Georgia Institute of Technology Microelectronics Center have been working on this problem by fabricating directional couplers that consist of two channels, or waveguides, placed closely together on a chip. Light entering one waveguide couples over to the adjacent guide. Researchers can control channel output by changing the index of refraction. Cascading a number of directional couplers creates an entire switching network.

Got rid of the electron problem? Noooo. The lithium niobate used to fabricate the couplers depends upon a changing electric field to vary the refractive index. So researchers decided to change materials.

To make an all-optical switch—free of electronic encumbrances—scientists chose aluminum gallium arsenide. Principal research scientist Chris Summers uses molecular beam epitaxy to grow quantum-well structures at the atomic level. A photolithographic masking technique then defines optical waveguides on top of the quantum wells.

Researchers are fabricating a directional coupler on a multiple quantum-well substrate to take advantage of the nonlinear optical characteristics of aluminum gallium arsenide. They hope to control the switching of two optical waveguides with an independent light source.

The ultimate plan is to install hundreds of thousands of the new devices into networks. Semiconductor lasers would furnish coupling control between adjacent waveguides to achieve switching speeds that approach the speed of light.

The remaining challenges include redesigning architectures to minimize cross-talk problems. [*For background information on quantum-well technology, see Micro View*, IEEE Micro, *Vol. 9, No. 1, Feb. 1989, p. 96.—Ed.*]

HDTV consortium may form

What are Apple, AT&T, IBM, Digital Equipment, Hewlett Packard, Motorola, Texas Instruments, and Zenith up to now?

The latest subject for consortium formation is high-definition television (HDTV). Each semiconductor superpower put up $50,000 to develop a consortium business plan that targets movie-print quality images for TV. High-definition images harbor implications for graphics, medical imaging, and electronic publishing. HDTV uses a high number of semiconductors to store and process TV signals.

Japan's national broadcasting company, or NHK, has been working on the idea since 1970, fueled by fees drawn from TV set owners and funds from the government. The Society of Motion Picture and Television Engineers in Hollywood has approved their system.

More to come on the news at eleven? Probably. The US government projects a $20-million market for HDTV within 10 years. Zenith, the only US maker of television sets, and AT&T have announced their joint proposal to the US Defense Department's Advanced Research Projects Agency for a $13 million award for technology to receive HDTV signals. The two companies separately submitted proposals for developing a display screen. DARPA plans to award $30 million in contracts this month.

Strategy for world communications

Cap International has formed a strategic planning service for the Integrated Services Digital Network. ISDN can simultaneously transmit voice, images, and information over one digital telephone line.

"ISDN may present the single largest opportunity for advancement in the communications market since Alexander Graham Bell invented the telephone," said the program's director, Michael A. Sprayberry. "Or it may present the single largest standards impasse as users, impatient for a carrier-provided solution, bypass the international standard and move toward highly intelligent, cost-effective proprietary solutions."

For more information, contact Martha Johnson, Cap International, One Longwater Circle, Norwell, MA 02061; (617) 982-9500.

Vaporware news

On January 17, the Federal Trade Commission announced a complaint and consent order against Coleco Industries based on charges that its ads falsely claimed that additional modules for its My Talking Computer were available for sale. The modules were not only not for sale but had not even been produced. The consent order prohibits any future misrepresentation of any computer-related products.

Benchmarks: speed versus cost

An independent laboratory specializing in performance benchmarking and architectural analysis of technical workstations reports some values for reduced instruction set computers (RISCs). Khornerstone-rating tests ranked eight workstations for speed in the following order: the Apollo DN 10000, MIPS M/2000-6 and M/120-5, HP9000/835, Sun 4/260 and 4/110, and IBM RT-135 and RT-125. (Khornerstone benchmarks rate CPU, floating-point, and disk-I/O performance.) The DN 10000 ran the race with over twice the speed of its nearest competitor (see Figure 1).

Egil Juliussen, Workstation Laboratories chair, explains that the DN 10000 performance is mostly due to its disk-subsystem organization and large memory size. He pointed out that the DN 10000 was three to four times faster than other RISCs in the disk I/O category—while its CPU speed was comparable to the MIPS M/2000.

But the laboratory was not content to let it rest at that level of analysis. They divided the list price of the workstation by the corresponding Khornerstone rating to arrive at a price/performance ratio called *dollars/Khornerstone*. The lower the number, the better the price performance. On that basis, the MIPS M/120-5 led the field, followed by the DN 10000. (See Figure 1 again.)

For more information, contact Phil Magney, ARS/Workstation Laboratories, 8111 LBJ Freeway, LB 156, Dallas, TX 75251; (214) 644-1733.

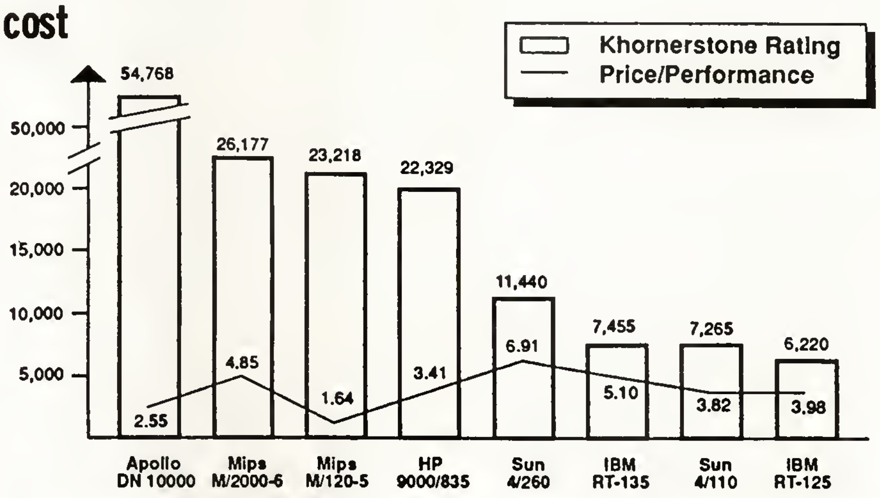


Figure 1. Khornerstone speed and price/performance ratings for eight RISC workstations. (Courtesy of Workstation Laboratories.)

ISSCC 89—New York City

James J. Farrell III
VLSI Technology, Inc.

[*IEEE Micro* welcomes well-written, succinct reports and impressions of industry conferences from attendees. Send contributions to Managing Editor, 10662 Los Vaqueros Circle, Los Alamitos, CA 90720; m.english (Compmail+).]

IEEE's International Solid-State Circuits Conference started on a chilly, wet Wednesday and ended on a cold but crisp Friday.

The ISSCC-rate, single hotel room cost over $160 per night, including tax. The conference preregistration fee (for IEEE members) was $170. On-site registration for non-IEEE members cost $220. Unless an attendee looked to find a moderate restaurant, food could get pretty expensive too. It was easy to spend $10 or more on breakfast. Guest parking at the hotel cost $24 a day. Conversely, New York City's taxicabs are moderately priced compared to those in most other large US cities. ISSCC in New York City is not an inexpensive event.

This was the first ISSCC in memory without Lewis Winner at the helm, handling the logistics of the conference. Lew died last year at the age of 83. The quality and continuing success of ISSCC is a fitting memorial to his efforts.

The conference featured a large number of excellent sessions and papers, most of which I could not attend because of time and other meeting obligations. However, I did manage to attend several interesting sessions, and I report my observations.

The first morning sessions started with data converters (which are getting much faster) and floating-point processors (which are getting both bigger and faster: 6.7 million floating-point operations). The floating-point processors included one with an 80-bit (that is a pretty big number) resolution from NEC and another promising 40 Mflops and 24 bits from Mitsubishi. As usual, the static RAM session presented some really lightning-fast devices that are getting pretty big—up to 1.0 megabits. An 0.8-μm, 2.0-nanosecond access-time SRAM from Hitachi looked like the speed champion of this session.

Following lunch, a tribute to Lew Winner, and presentation of several well-deserved awards, Heitaro Nakajima of Sony presented an excellent tutorial on the history and future of digital audio. To support his talk, Nakajima had three recorded presentations to demonstrate the effects of varying word size, sampling rates, and data compacting techniques. Finally, Nakajima made a very compelling case for the very widespread acceptance of compact disk technology, not only for digital audio but for digital video and computer memories.

The microprocessor session was mobbed. Four of the five papers on microprocessors presented RISC architectures. Speed (up to 50 MIPS peak) and size (64 bits) were the main topics. Although not in the printed program, Intel briefly spoke of its new 64-bit offering, the N-10. This device not only contains the RISC core but instruction and data caches, floating-point multiplier, bus control, and 3D graphics. Intel representatives said that they saw 50-MHz samples on the first silicon. With a million transistors on board, three watts of power dissipation doesn't seem unreasonable, but it is surely enough to keep its 168-pin grid array package fairly warm unless it has a heat sink. I think it is fair to assert that RISCs dominated this session.

Other sessions included those on imagers and sensors, high-speed BiCMOS, nonvolatile memories, data communication, gate arrays, video and image processors, analog processors, telecommunications, high-speed digital circuits, and a well-rounded selection of diverse topics discussed during the informal evening sessions.

The concluding day of the conference held what for many attendees is the mainstay of ISSCC: sessions on dynamic RAMs. The DRAM session continues to be a good indicator of both the progress in semiconductor technology and future trends. One paper, appropriately from Krysalis, introduced a 16-Kbit ferroelectric, nonvolatile RAM. If this technology proves viable, we could be seeing another significant memory trend.

Mitsubishi and NEC representatives discussed two 16-Mbit DRAMs, and Hitachi people talked about a 1.5-volt DRAM for battery-operated applications. The 200-Mbit (yes, 200-Mbit) wafer memory from Fujitsu was interesting enough to attract the interest of *Business Week*, which reported on the device. However ambitious and appealing this much memory is, it may yet be a few years before we start to mount six-inch wafers directly onto the printed circuit board in high volume.

In conclusion, I felt integrated circuit technology was moving forward briskly. Submicron designs seem universally well underway. Access and cycle times are headed for the low nanoseconds, while remaining in silicon technology. However, 5-volt-only systems may present a problem to system architects. Many submicron devices must operate with 3.3-volt power to avoid voltage stress failure. Some convert 5 volts to 3.3 volts on chip, but others require 3.3 volts from the system. It may not be a "5-volt-only" world too much longer.

Reader Interest Survey

Indicate your interest in this department by circling the appropriate number on the Reader Interest Card.

Low 183 **Medium** 184 **High** 185

Continued from p. 10

derivative work to its customers. Further, it might state that its customers did not make copies of a derivative work, since the customers never had a tangible copy of the alleged derivative work. Unless customers printed out the screen displays or contents of RAM, probably no "copy" of any derivative work would have existed, as the Act defines a copy. A copy is a material object in which a work is fixed and from which the work can be perceived or reproduced. A work is "fixed" only when its embodiment in physical form is sufficiently permanent or stable that it can be perceived or reproduced "for a period of more than transitory duration." Thus, RAM is arguably impermanent, so that no copy exists.

On the other hand, the statutory provision protecting derivative works, section 106(2) of the Copyright Act, simply gives copyright owners the exclusive right to prepare derivative works based on the copyrighted work. There is no mention of "copies." The legislative intent, in regard to works (such as ballets, pantomimes, and improvised performances) that are performed, rather than reproduced and sold as such, was that copyright infringement for preparing such a work (or, for that matter, performing or displaying it publicly) could be found without making a tangible copy—an act that might never occur. The final result embodied legislative overkill, for the language was not limited to pantomimes and the like. Instead, it is broad enough to cover programs and screen displays.

To make matters even less certain, Congress failed to define "prepared." Therefore, "to prepare a derivative work" could, in principle, mean simply imagining it, so to say, working it out in one's mind and speaking of it. Just as one could be a felon for imagining the King's death, perhaps one could be a criminal for imagining a derivative work.[5]

Obviously, there must be some limit to the concept of "prepare," but equally obviously, there is considerable room to argue that preparation of the derivative-work version of a computer program or screen display in RAM or on the screen of a monitor is actionable as copyright infringement. By the same token, causing one's customers to do these things may be contributory infringement.

Thus the expansive concept of derivative work put forward in the *Mirage* and *Artic* cases could pave the way for holding creators of add-on computer programs to be copyright infringers, even though they never made or marketed computer programs containing any code appropriated from the original computer programs. Is this a sensible direction for copyright law to take? Is there a rationale for resolving the copyright liability issue here?

The decision in the *Artic* speed-up kit case comes closest to articulating a rationale for copyright liability. But its rationale is vacuous. The question is who is to get the surplus value. It is not a persuasive answer for the court to state that the copyright owner should get it because he "would undoubtedly like to lay his hands on some of that extra revenue." This must be the King David and Uriah's wife theory of copyright; or Willy Sutton's explanation of why he went to banks. (That is where they keep money.) The principle proves too much and therefore is not legally acceptable; at least it would not be acceptable in fields other than copyright law.

The expansive concept of derivative work put forward in the Mirage and Artic cases could pave the way for holding creators of add-on computer programs to be copyright infringers.

The underlying fact situation in all of these cases, and in the add-on software case as well, is this: Copyrighted goods are sold for a price that reflects the copyright owner's limited knowledge of future developments in the marketplace and in technology. A later entrepreneur discovers another market niche, or develops some technology that permits the creation of another market niche, for the goods. The niche is worth money—call it surplus value or extra revenue. There are two principal claimants for the right to the revenue: the copyright owner and the purchaser of the copyrighted goods who desires to exploit the new niche. We may call them the original entrepreneur and the subsequent entrepreneur. There is also the public, somewhere in the background, but theoretically the real beneficiary of the copyright system.

The surplus value can be allocated by the State—that is, by the copyright laws as interpreted and administered in the courts, since Congress has not expressly addressed the point in the copyright laws. The State can allocate the surplus value either to the original or subsequent entrepreneur, or it can divide it between them in some way. The question is which of these allocation choices best effectuates whatever policies the copyright laws are supposed to accomplish.

According to the Constitution, intellectual property law exists to promote the progress of science and useful arts. This in turn involves a difficult balance between the interests of authors in the control of their works on the one hand, and society's competing interest in the free flow of information and technology. Presumably, the long-term interest of the public predominates over other interests. In this context, the long-term interest of the public is maximizing, in the long run, the creation and availability to the public of technological advances. Reward to authors is instrumental to this end; it is not an end in itself.

Does allocating the surplus value of the new technological use to the original or subsequent entrepreneur lead to more creation and marketing of technological advances?

The answer is probably that society gains more in the long run from rewarding the subsequent entrepreneur than the original one, in respect to the surplus value created by the subsequent entrepreneur's new use. By hypothesis, at the time of creation of the original and its marketing, the original entrepreneur does not foresee the add-on technology, although the entrepreneur may generally suspect that some kind of improvement is inevitable, eventually. Hence, the discounted present value of the potential revenue from the unforeseen add-on technology is not much of an additional incentive to the original entrepreneur.

On the other hand, the subsequent entrepreneur with a specific idea for an add-on—for example, an enhancement of a spreadsheet—has to decide whether it will make enough money from the add-on to persevere in trying to develop and market the add-on. If the law is that the subsequent entrepreneur is liable to be shut down at the will of the original entrepreneur, because the latter is legally entitled to control use of the add-on as a derivative work, then that will be a powerful disincentive to the subsequent entrepreneur.

Society gains more in the long run from rewarding the subsequent entrepreneur than the original one.

In fact, it may make no sense at all to persevere. At the very least, the threat of facing a countervailing monopoly is a strong disincentive. The bargaining power of the two entrepreneurs is quite unequal, for the subsequent entrepreneur cannot shut down the original entrepreneur in its exploitation of the original program. Here, the monopoly power would work mainly in one direction: down the distribution chain. The result of treating add-ons, as involving derivative works, therefore, is likely to be that creation and marketing of such technology is decreased relative to that under a contrary legal regime.

I referred to the possibility of dividing the surplus value between the two entrepreneurs. In theory, intermediate solutions typically yield greater returns than those at extreme values of the parameters. This might suggest a regime under which the later entrepreneur would pay a royalty or share of the profits to the earlier entrepreneur. Some Ninth Circuit precedents suggest such an approach. But it is doubtful, or in any event quite unclear, that this rationale could be extended to software add-on programs and similar derivative-work disputes. First, no other circuit has shown the Ninth Circuit's enthusiasm for this kind of judicial creativity or activism. Second, it is one thing to patch up an existing controversy by selecting the lesser evil, which is to give the copyright owner a monetary award based on the relative contribution to profits attributable to the infringed work. But it is quite another thing to institute a regime for doing this in future cases that have yet to arise.

The Ninth Circuit's decisions involve cases in which the parties had a reasonable difference of opinion about their respective legal rights, and the only way to resolve the matter was for a court to adjudicate it after the event. But it would be a farther and bigger step for a court to impose reasonable-royalty licensing, and deny any injunction, in a case where the plaintiff had an unquestioned right under the copyright law. I do not say that the result is not a better social or economic one, but decreeing it seems more a legislative than judicial prerogative. I would expect most courts to be too diffident to take this step. Moreover, absent an industry consensus on the point (which probably does not exist), I would not expect Congress to legislate such a measure. Accordingly, it appears that only one of the two winner-take-all legal options is available in regard to add-on programs.

Perhaps, in the *Mirage* case it does not matter who wins. It may make little difference to society whether Albuquerque or Mirage profits from framing Nagel's art or pasting it onto tiles. Similarly, it probably makes no difference one way or the other whether it is held that a derivative work is created when a moustache is painted onto a copy of the Mona Lisa. And the *Artic* court may have rightly said "Who cares?" about running 33-rpm records at 78 rpm. But the same court said that many people cared about the revenue from video-game speed-up kits. It then proceeded to rule in a way that would discourage any more speed-up kits from being created. Perhaps that, too, does not matter.

But if add-on programs were subjected to the same rule, that would be a significant setback to technological advancement in software and a misfortune for software users. I believe that it is fair to say that society will lose considerably if the creation and marketing of add-on programs are suppressed in the name of a derivative-work rule. The better rule would be that add-on programs are not derivative works, and that their use by purchasers of software does not involve preparation of derivative works.

Given the ambiguity of the statue and its inconclusive legislative history, the courts are surely free to give the term *derivative work* a scope that is more, rather than less, consistent with carrying out the fundamental goals and purposes of the copyright system of promoting progress in the field to which it is applied. That scope would be to interpret a derivative work not to include an add-on program, as such, unless it incorporated substantial enough portions of code from the original program to involve reproduction, at least in part, of the original. In the same vein, use of an add-on program should not be considered preparation of a derivative work, or alternatively should be a privileged use of that which the original seller sold to its customers.

References

1. In late 1988 Ashton-Tate, publisher of the dBase II and III programs, sued Fox Software and The Santa Clara Operation for copyright infringement, alleging that the defendants' programs unlawfully use copyrighted subject matter taken from the dBase programs. The plaintiff's legal theory is unclear. Among other things, the plaintiff's president has asserted that the defendants are liable to the plaintiff for utilizing the dBase programming language, in which the plaintiff asserts a copyright. There also appear to be look-and-feel claims.
2. The term *image* is used in the Coypright Act to define an audiovisual work and its performance. It is not clear whether the protected images of an audiovisual work are each of the entire screens or each of the individual separate components of the screens.
3. *Artic Int'l, Inc. v. Midway Mfg. Co.,* 704 F.2d 1009 (7th Cir.), *cert. denied,* 464 US 823 (1983).
4. See, e. g., *Digital Communications Assoc., Inc. v. Softklone Distrib. Corp.,* 659 F. Supp. 449 (N.D. Ga. 1987); ***Broderbund Software, Inc. v. Unison World, Inc.,*** 648 F. Supp. 1127 (N.D. Cal. 1986).
5. Willful copyright infringement for commercial gain can be a felony or misdemeanor. Felony conviction requires reproduction or distribution of more than a specified number of copies of certain types of work.

Reader Interest Survey

Indicate your interest in this department by circling the appropriate number on the Reader Interest Card.

Low 171 **Medium** 172 **High** 173

New Products

Marlin H. Mickle
University of Pittsburgh

Send announcements of new microcomputer and microprocessor products, and products for review, to Managing Editor, IEEE Micro, *10662 Los Vaqueros Circle, Los Alamitos, CA 90720-2578.*

The business of RISC

Sparc or MIPS: Pick a chip set

LSI Logic has implemented two two-chip sets based on Sun Microsystems and MIPS RISCs.

The 25-MHz Sparc L64801 and 33-MHz L64811 each offer two processing units, one 4-Gbyte memory management unit, and a floating-point controller. Overlapping register windows minimize the number of load and store instructions.

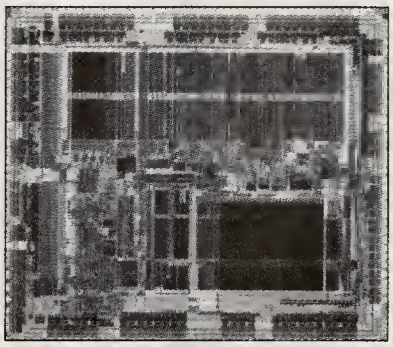
The L64801 integer unit incorporates a 120-register file with seven overlapping windows into a 32-bit architecture. The external chip interface consists of separate 32-bit address and data buses, along with a 32-bit coprocessor interface for an optional Texas Instruments 74ACT8847 floating-point unit.

The L64811 integer unit furnishes 136 registers with eight overlapping windows and a second coprocessor interface.

The MIPS-based LR2000 and LR3000 feature a MIPS instruction set, 32-bit instructions with single addressing mode, and register-to-register load and store operations. With the exception of Multiply and Divide, instructions execute in one cycle. Speeds range from 12.5 to 16.7 MHz for the LR2000 and from 16.7 to 25 MHz for the LR3000.

The company lists development tools for both sets of RISCs and plans application-specific IC versions in 1989. **LSI Logic.**

L64801 **Reader Service Number 11**
L64811 **Reader Service Number 12**
LR2000 **Reader Service Number 13**
LR3000 **Reader Service Number 14**



The company states that the third-generation LR3000 RISC in this photomicrograph delivers four times the performance of the original R2000 MIPS.

Military R3000 takes orders

The military version of the IDT79R3000 RISC operates at 16.7 MHz with a sustained rate of 12 MIPS. Fabricated in IDT's enhanced CMOS (CEMOS) technology, the processor features 0.8-micrometer gate lengths and ceramic PGA packaging. Sample quantities are now available; production quantities are planned for October. **Integrated Device Technology; $720 (100s).**

Reader Service Number 10

BCS open for business

The 88000-based Icon 8000 system serves up to 256 business users with 15 MIPS of performance. Complying with the 88000 Binary Compatibility Standard developed by the 88open Consortium, the architecture allows software to operate transparently across systems. BCS, based on AT&T's Unix System V Version 3.0, combines X-Open and IEEE standards. **Sanyo/Icon Int'l.**

Reader Service Number 15

Acorn puts it in a nutshell

The 32-bit VL86C020 Acorn RISC features a general-purpose CPU and a full cache memory subsystem within the same device. This strategy decouples the clock from the memory system and lowers processor bandwidth memory demands. Other features include upward software compatibility with the VL86C010 at over two times its performance and an added Semaphore instruction for multiprocessor support. The instruction performs an indivisible read-modify-write cycle to memory to manage globally allocated resources. Available in 16- or 20-MHz versions, the machine uses typical DRAM devices.

The CMOS-implemented device for laser printers, graphics engines, and network protocol adapters comes in 160-pin, plastic quad flatpack packages (PQFPs). **VLSI Technology.**

16 MHz **Reader Service Number 16**
20 MHz **Reader Service Number 17**

Chip set melds 50 components

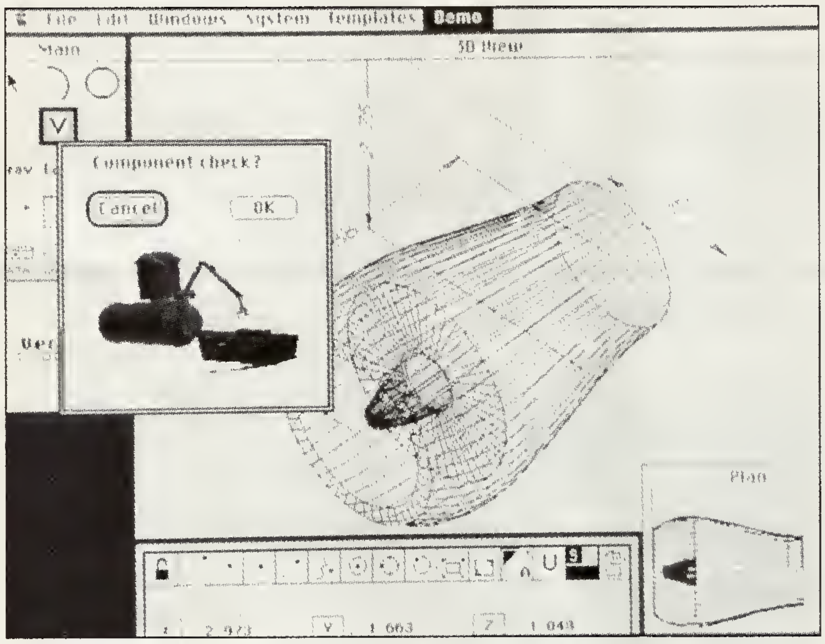
The 88000 RISC family combines an 88100 CPU chip that performs integer and floating-point execution with an 88200 MMU chip that contains separate data and instruction caches. The company states that the tandem chips provide the same functions as 50 or more components in competing systems.

The 20-MHz system attains performance ratings equivalent to up to 17 VAX MIPS. **Motorola; $494 (88100); $619 (88200).**

88100 **Reader Service Number 18**
88200 **Reader Service Number 19**

CAD and graphics on the Macintosh II

Forget learning Fortran



Vector software lets the developer create and link Hypercard-like buttons to develop interactive designs.

The Vector engineering-applications generator allows users to create specialized software for niche applications without learning Fortran or C programming languages. The package also provides an open-architecture geometric database that can be modified to a specific application.

Vector includes complete source code for 3D, including associative dimensioning, spline curves, and text manipulation. Built-in macros help create 3D wireframe models. Users may define multiple views, view models from any direction, zoom and pan the display using any scale, measure and change elements of the database, or define and parametrically adjust the model. A spreadsheet-like interface links geometric dimensions to numeric data and formulas. **Microconcepts; $2,595.**

Reader Service Number 21

Synthetic camera renders 3D

Mactracer software allows users to create a digital synthetic camera and generate realistic 3D imagery without having to build the actual model. Files can be saved in Pict or TIFF formats for printing to film recorders, laser printers, videotape, or pasting into color graphics programs. Image synthesis features include shadow casting, reflections, refractions, compositing, and multiple levels of antialiasing. A modeler contains utilities for hierarchical data structures, surfaces of revolution, and extrusion. **Ray Tracing Corp.**

Reader Service Number 20

CAD bridges design disciplines

The Dreams CAD software package offers MacDraft-like geometric drawing tools, Bezier and spline curves, parallel lines, and freehand sketching. Features include zoom, rotation of text and objects in fractional degrees, custom-line styles, and associative dimensioning. Users can store commonly used symbols, details, and drawings in on-line libraries.

Applications include architecture, engineering, and construction and desktop-publishing markets. **Innovative Data Design; $500 (retail).**

Reader Service Number 22

Mainframe analysis comes to the PC

The Cosmos/m finite-element-analysis program for the Macintosh II breaks down geometric objects into 25,000 degrees of freedom (elements). Using substructuring, the program solves a 100,000-degree-of-freedom system. Color-filled stress and thermal-contour plots employ 256 colors.

An advanced Cosmos/m package solves nonlinear problems by means of an incremental solution technique and uses isoparametric elements to increase accuracy in cases of curved boundaries or high-stress variations. This custom-written package also offers nonlinear buckling analysis previously confined to mainframe programs.

The program requires 4 Mbytes of RAM and a 50-Mbyte hard disk. **Structural Research and Analysis Corporation; $1,995 (linear, static, and dynamic analysis package).**

Reader Service Number 23

Process color on the Mac

The Photomac photodesign software for the Macintosh II features design tools for retouching, montaging, masking, resizing, and rotating images. Color-prepress features include red, green, and blue color correction, brightness/contrast adjustment, smoothing, and sharpening. The software handles images scanned at 24 bits per pixel and gives the user a palette of 16.8 million colors.

Photomac-generated, four-color separations print on Postscript printers and imagesetters. The user can set screen rulings, screen angles, and separation dimensions. Adjustments for paper type, gray-component enhancement, and gray balance give added control over each separation. Images can also be printed on color printers and film recorders.

The software requires 2 Mbytes of RAM, a 40-Mbyte hard disk, and an Apple 8-bit video card. **Data Translation; $695.**

Reader Service Number 24

SCSI connections

Controller reaches 20 MHz

The WD33C93A SCSI bus interface controller boasts 16- or 20-MHz versions, combinational commands, and a 5-Mbyte/s data-transfer rate. It executes arbitration, disconnecting, reconnecting, parity, and synchronous data-transfer protocols. Multiphase SCSI sequences execute under control of its own internal microcontroller. The WD33C93A can function as either initiator or target, as part of the host computer system, or as part of an intelligent peripheral.

Advanced Micro Devices has incorporated the devices into the 16-MHz Am33C93A-16JC and the 20-MHz Am33C93A-20JC controllers. **Western Digital or Advanced Micro Devices; $15.60 (16-MHz) (1,000s); $18.20 (20-MHz) (1,000s).**

16-MHz Western Digital **Reader Service Number 25**
20-MHz Western Digital **Reader Service Number 26**
16-MHz AMD **Reader Service Number 27**
20-MHz AMD **Reader Service Number 28**

Two chips boost buffering

The ESP 100A and 200 small computer systems interface chips support DMA buffer-memory interfacing at 12 Mbytes/s.

The company states the 100A is pin compatible with the original ESP chip. The 200 lets the parity bit pass through directly from the SCSI bus to the local data bus (host or peripheral). **Emulex; $19.50 (either chip, 1,000s in May).**

100A **Reader Service Number 31**
200 **Reader Service Number 32**

Once is enough

The Fasst interface open-architecture software lets developers use just one set of commands to talk with a SCSI interface. Fasst can be used in MS-DOS, Windows/386, Novell Netware, Unix, Xenix, PC-MOS/386, and OS/2 operating systems. A set of "software registers" replaces the need for hardware-register compatibility. **Western Digital.**

Reader Service Number 34

Multibus-II adapter adds firmware

The CD22/4550 SCSI host adapter features an on-board, 12.5-MHz 80186 CPU and 512 Kbytes or 2 Mbytes of parity-protected DRAM for data buffering and high-speed disk caching. The board also provides an SBX connector for one 8- or 16-bit module and is compatible with SCSI/ANSI X3T9.2 single-ended or differential interfaces. An 82258 advanced DMA controller delivers SCSI data-transfer rates of 1.5 Mbytes/s (asynchronous) and 4 Mbytes/s (synchronous).

Fastcache firmware makes the adapter compatible with the Intel 186/224A disk and tape controller and performs disk-sorting, read-ahead, write-back, and disk-caching functions. **Central Data; $2,255 (100s).**

Reader Service Number 35

Keeping an eye on the network

The Local Area Network Camera Control System for video surveillance and telemetry control allows view and control of up to 255 video cameras on 16 monitors. Contact-closure bus devices alert operators to a disturbance and immediately direct the camera to that location. **C-Cor Electronics.**

Reader Service Number 29

Is your protocol up to snuff?

The Sniffer protocol analyzer diagnoses 16-Mbit IEEE 802.5 Token Ring LANs built around the TMS380C16 chip set. The Sniffer hardware and software system monitors, collects, and decodes network protocol information. **Network General; from $15,750.**

Reader Service Number 30

OS-9 passes tokens to ARCnet

The Variable Block File Manager and Network File Manager device-driver software connect systems running the Microware OS-9 operating system to the token-passing ARCnet networking structure. The VBF/ARC provides a low-level interface to the network with high-speed data transfers between systems. The NFM/ARC features the same functions as an OS-9 disk file manager. **C&C Technology; from $125 (license, per copy) (VBF/ARC); from $160 (license, per copy) (NFM/ARC).**

VBF **Reader Service Number 36**
NFM **Reader Service Number 37**

CD-ROM holds 680 Mbytes of data, stereo sound

With an average access time of 400 ms and a 2.75 × 8 × 9-inch size, the XM-3201A CD-ROM disk drive supports desktop and portable computer applications. The stand-alone subsystem incorporates a 680-Mbyte drive with controller, SCSI connector, power supply, and external drive packaging. The drive's integral audio capability allows high-fidelity stereo sound to accompany stored text and images. **Toshiba America; $1,095.**

Reader Service Number 33

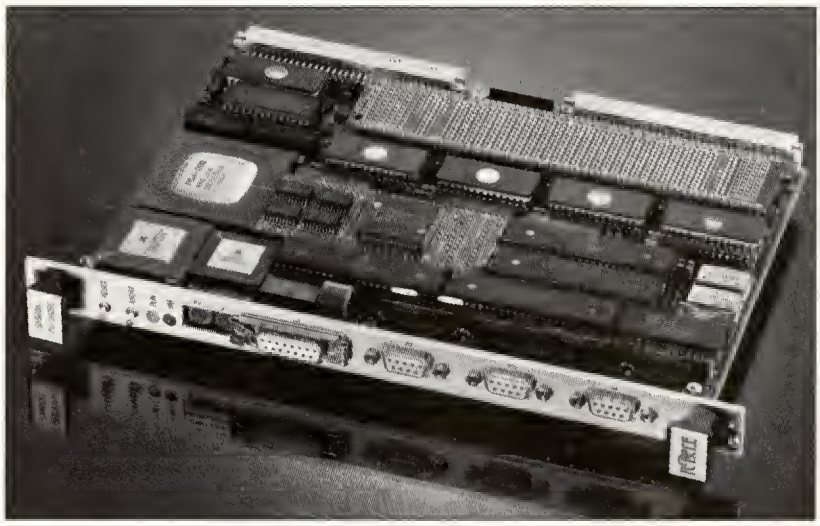
Fiber-optic network integrates SNMP

According to the company, the p4200 Router is the first multiprotocol internetwork router to implement the Simple Network Management Protocol (SNMP) standard. It combines with the Pronet-80 token-ring network to integrate diverse LANs such as Ethernet, Token Ring, and wide-area links across a high-speed backbone. **Proteon.**

p4200 **Reader Service Number 38**
Pronet-80 **Reader Service Number 39**

Single boards

CPU talks with VMEbus



Four multiprotocol serial ports in the CPU-30 provide RS-232 or RS-232/422/485 compatibility.

The CPU-30 features a 68030 microprocessor (in 20- or 25-MHz versions), a 68882 floating-point coprocessor, and a 32-bit DMA controller that transfers data at high speeds locally and across the VMEbus. A 22,000-gate, 281-pin application-specific IC enables multiprocessing via message passing, mailbox interrupts, and a comprehensive VMEbus interface.

An 8-bit-wide, 8-byte-deep FIFO message buffer with an 8-bit-wide, high-priority message register enables a scheme of 256 possible messages. Up to 20 VME/Plus-equipped boards can receive a message simultaneously. Eight software-driven mailbox interrupts augment seven hardware interrupt levels. **Force Computers; $6,450 (1-9 boards) (20-MHz 68030, 68882, 4-Mbyte memory, floppy and SCSI controllers); $7,250 (25-MHz version with Ethernet controller as well).**

20 MHz **Reader Service Number 41**
25 MHz **Reader Service Number 42**

Versatility supports DSP

Two DSP/AIB boards for multichannel analog data-acquisition support real-time DSP or control applications in the DSP5600ADS. Both DSP/AIBs offer two channels of analog input, one channel of analog output, and serial and parallel digital I/O. Users can program a variable sampling rate of up to 20 KHz (single channel) plus 10 bits of linearity over any 10-bit portion of a 14-bit dynamic range.

The DSP/AIB-1 performs 56000ADS-data acquisition through a provided serial cable. The DSP/AIB-2 does the same or connects to a PC backplane. The -2 fits into a half-card IBM PC XT/AT slot and also performs PC acquisition. Both boards include 56000 software drivers. **Sonitech Int'l.; $495 (DSP/AIB-1); $695 (DSP/AIB-2).**

DSP/AIB-1
Reader Service Number 44
DSP/AIB-2
Reader Service Number 45

Multibus-1 migrates to Eurocard

The GESMPU-18A 16-bit, 80286-based microprocessor provides 512 Kbytes of fast-access, on-board dynamic memory and sockets for 256-Kbyte EPROMs on a single-height Eurocard. If needed, an 80287 arithmetic coprocessor can extend directly to the 80286 instruction set. Features include two RS-232 serial ports, three timers, and a clock calendar with battery backup. The processor's MS-DOS operating system allows it to be used as the core of an IBM PC AT system on the G-64 bus. **Gespac; $1,450.**

Reader Service Number 40

AT-sized card holds it all

The QC 1094 80386SX-based processor accesses up to 8 Mbytes of on-board RAM while running at 16 MHz. An 82335 memory-management chip allows zero-wait-state operation while the 100-nanosecond RAM minimizes system cost.

The board plugs into any IBM AT-compatible passive backplane and operates at 8 Hz to avoid any potential peripheral incompatibility. It comes with interfaces for floppy and hard disks as well as two serial ports and one printer port. **Quam; $895 (OEM discounts).**

Reader Service Number 43

Fast board joins Multibus II

Based on a 20 MHz-80386 microprocessor, the CD22/1386 computer for Multibus II applications is hardware and software compatible with the Intel 386/120, according to the company.

It has two SBX connectors, features 4 Mbytes of dual-port dynamic RAM, and employs four double-word, interleaved banks of page RAM to achieve zero wait states for most accesses. The CD22/1386 also contains an 82258 DMA controller for solicited message transfers and I/O control. A flyby mode enables two DMA channels to transfer messages between the Multibus II message-passing coprocessor and DRAM at 13.3 Mbytes/s. **Central Data; $6,225 (4 Mbytes of parity-protected RAM) (100s); $5,240 (4 Mbytes without parity).**

Parity MHz
Reader Service Number 46
No parity MHz
Reader Service Number 47

Users plug in DSP

The DSP Link family of modular, PC-based hardware incorporates single-chip DSPs from Texas Instruments, Motorola, and Analog Devices. Users can choose from a number of data-acquisition and control peripherals. These peripherals provide real-time DSP with the following options: 4-channel analog I/O, 32-channel analog input, DSP Link prototyping, 16-channel analog output, pro-audio boards (two), and dual-processor communications. **Spectrum Signal Processing; $845 (I/O); $845 (32-channel input); $95 (prototyping); $845 (16-channel output); $1,450 or $1,950 (boards); $95 (communications).**

I/O
Reader Service Number 48
32-channel
Reader Service Number 49
Prototyping
Reader Service Number 50
16-channel
Reader Service Number 51
Boards
Reader Service Number 52
Communications
Reader Service Number 53

Teach 68000 programming on a PC

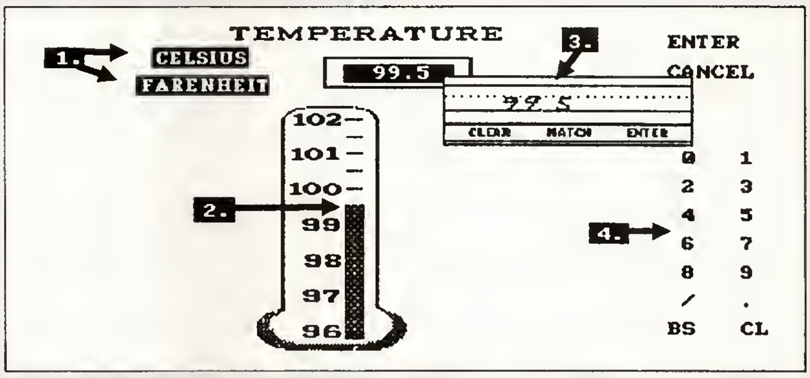
The Mister-8 computer system for teaching 68000 programming plugs into either an IBM PC or AT. Because the board uses a 68008, mainframe solutions are unnecessary to run and debug programs. Motorola's Tutor monitor provides prompt feedback. For more complex projects, users can employ the PC editor and cross-assembler tools to write and document programs. The Mister-8 monitor provides a suite of system-call routines for program development in both C programming and assembly languages.

Micro Resources; $345 (board with monitor and serial port); $75 (Motorola's Tutor monitor EPROM); $15 (modification disk for Tutor).

Board **Reader Service Number 54**
Tutor **Reader Service Number 55**
Disk **Reader Service Number 56**

Across the product line

Finally: Just write on the screen!



The Code-write software-development kit allows C-language programmers to create or modify programs. Graphic objects offer freehand or computer-aided line-drawing generation and editing in the field.

The Linus Write-top portable computer system recognizes and converts noncursive handwriting into ASCII when a user prints on a flat LCD screen with a mechanical stylus. An adaptive symbol-recognition algorithm learns handwritten symbols, including scientific notions. Users can incorporate hand-drawn sketches and input data while walking or standing.

Forms-write/Arrow software allows users to collect data for real-time calculations, build a database, or transmit data over an optional modem. The 9-lb, battery-operated desktop has serial RS-232C communications, printer, and IBM PC XT-compatible keyboard ports. Recognition training is mandatory. **Linus Technologies.**

Write-top
Reader Service Number 57
Forms-write
Reader Service Number 58
Code-write
Reader Service Number 59

Moving?

Address changes: Please notify us 4 weeks in advance

ATTACH LABEL HERE

- This address change notice will apply to all IEEE publications to which you subscribe.
- List new address above.
- If you have a question about your subscription, place label here and clip this form to your letter.

Name (Please Print)

New Address

City

State/Country Zip

Mail to:
IEEE Micro, Circulation Dept.,
10662 Los Vaqueros Cir.,
Los Alamitos, CA 90720-2578

New Products

AT masters Q-bus systems

The 405 IBM PC AT-Q22 bus adapter makes the IBM PC AT appear as a bus master processor on the Digital Equipment Corporation Q-bus. This interface results in AT processing power, I/O features, communications, and software support in PDP/LSI and MicroVAX Q-bus applications. The adapter consists of one printed circuit card that fits inside the AT and another one that fits inside a Q-bus card cage. The AT can serve as one of several bus masters in a multiprocessor application. Both byte and word transfers are supported. The adapter also works with the IBM RT, Compaq 286/386, Apollo 300/4000, Sun 386i, and AT compatibles. **Bit3 Computer Corp.; $2,495.**

Reader Service Number 60

Sensor expands to 4 million pixels

According to the company, its 4-million-pixel KAF-4200 electronic image sensor has twice the resolution of other sensors on the market. Each pixel senses impinging light and converts that light into electrons, which are processed to produce a video signal representing the image. A pixel measuring 9 × 9 micrometers falls into an array of 2,048 vertical columns and 2,048 horizontal rows. This monochrome system component for still-image capture is a silicon charge-coupled device.

Logic and signal-processing circuitry consists of two board sets. The primary board contains the logic functions for the timing signals, the sensor bias, and the driver circuits. The sensor is mounted on a secondary board which also contains the sample-and-hold circuits and video amplifiers that drive a 75-ohm cable. **Eastman Kodak; from $65,000.**

Reader Service Number 61

AI scrutinizes disks

The technical skills needed to run Disk Technician Advanced hard-disk preventative maintenance are embedded in its software. Artificial intelligence pattern recognition seeks out problem signs, while an expert system determines appropriate testing and repair routines.

DTA tests and repairs drives in one run and performs automatic reinterleaving and system/data recovery. Database storage of test history helps the AI system detect difficult random and intermittent errors. Any discovery of marginality or error prompts the DTA to write a new track, which is then tested. The program can run from a work copy, be copied between 5.25- and 3.5-inch formats, installed on a hard drive, or transferred to another PC via modem or LAN. **Prime Solutions; $189.95.**

Reader Service Number 62

Smile when you say that, virus

The Protec Version 3.3 security system boot-protects hard disks with DOS 4.00, DOS 4.01, and MS-DOS 3.3-R operating systems that have partitions between 32 and 300 Mbytes. Software boot protection guards against intruder access to the hard disk from a floppy. A log-on identification/password mechanism employs user-defined password lengths and password aging. If necessary, an automatic feature segregates users from sensitive files while allowing them to share a program.

The anti-software-theft system stops users from copying licensed software off the hard disk. DOS-level file-execution control stops users from issuing certain commands and programs. After an enhanced virus-detection module injects a guardian virus, an audit trail reports any violations and system-usage data. **Sophco; $195.**

Reader Service Number 63

Sharing a piece of the Apple

The uShare network system uses Appleshare software to allow file sharing between a Macintosh and host Sun workstation. The uShare network provides VT100 terminal emulation, Unix-compatible electronic mail, and print spooling. **Information Presentation Technologies; $1,195 (Sun host); $149 (Appleshare software).**

Host **Reader Service Number 64**
Software **Reader Service Number 65**

Servers support Concurrent DOS

Compupro network servers for the IBM PC family and compatibles in Digital Research networks come in 80-, 150-, 300-, and 600-Mbyte internal hard-disk-drive sizes. All servers include Concurrent DOS, DR Net 2, ARCnet interfaces, and a 512-Kbyte hard-disk cache that supports 12 Mbytes/s data-transfer rates. **Compupro; from $6,500.**

Reader Service Number 66

RAM supports 80386 processors

Directly interfacing to the Intel 82385 cache controller of the 80386 32-bit microprocessor family is the V63C328 8K × 16-bit cache data RAM. A built-in mode control lets users of the CMOS device configure the memory internally as either one 8K × 16 or two 4K × 16 devices through the use of a programmable pin. In a two-way set associative cache, two V63C328 chips replace 26 other devices, including fast SRAMs and glue logic, to save 8 square inches of board space. The V63C328 in a 52-pin PLCC is available in small quantities. **Vitelic Corporation; $100.**

Reader Service Number 67

Reader Interest Survey

Indicate your interest in this department by circling the appropriate number on the Reader Interest Card.

Low 180 **Medium** 181 **High** 182

Product Summary

Marlin H. Mickle
University of Pittsburgh

For more information, circle the appropriate Reader Service Number on the Reader Service Card.

MANUFACTURER	MODEL	COMMENTS	R.S. NO.
Mass storage notes			
Mountain Computer	Mac Filesafe 150 backup system	Streaming-tape subsystem for the Macintosh SE and II stores 150 Mbytes of data with an image-backup speed of 6 Mbytes per minute in a QIC-150 format. The 12-pound unit fits into a half-height, 5.25-inch form factor and uses the DC600XTD quarter-inch cartridge. $2,195.	80
3M Data Products	DC2080 quarter-inch cartridge	Data cartridge features 80 Mbytes of storage on a 3.5-inch form factor with an Irwin recording format. Shipped with IBM PS/2 tape-drive systems, the unit has media ratings of 90 ips with 205 feet of tape at a density of 10,000 bpi.	81
Memorex Computer Supplies	Color disks	Ten-disk packages offer blue, green, yellow, orange, and red color coding of subjects or applications for increased organization of disk-filing systems. The disks come in either 3.5- or 5.25-inch sizes. The company also provides packs of 50 5.25-inch 2S/2D flexible disks.	82
Maxell Corporation of America	MC-6000 XHP data cartridge	Tape-backup cartridge measuring 4 × 6 × 0.665 inches features a 150-Mbyte recording capacity for one-quarter-inch tape drives. The 620-foot tape is compatible with 3M's DC-600 XTD cartridge drive and IBM's AS-400 backup system.	83
Xylogics	772 controller	One-half-inch-tape controller resides on a VME board and provides DMA speeds of 10 Mbytes/s and data-transfer rates of 1.25 Mbytes/s. The 772 can run up to eight one-half-inch tape drives and handles GCR tape transports.	84
Computer Spectrum	Diskplus PCB	Board fits into cabling between the b: disk-drive and its connector to support a 3.5-inch, third-floppy drive without occupying an expansion slot. Can provide internal or external drives with 720-Kbyte or 1.44-Mbyte storage. Board doubles as a backup unit. From $99.	85
Colorado Memory Systems	QFA-500 tape drive	QIC-150 drive uses 2-to-1 data compression and a 1,000-foot-tape-length to offer a 500-Mbyte capacity in either internal or external configurations. Also offers a 1×10^{-15} bit error rate and compatibility with IBM PCs and PS/2s. From $1,395.	86
Maxell Corporation of America	MF2-ED and MF2-TD magnetic disks	Two 3.5-inch floppies service 4- and 12-Mbyte-range drives with 35,000- and 36,700-bpi performance. Using barium ferrite or metal powder as recording materials, the floppies flaunt 1,000- and 1,250-Kbyte data-transfer rates and 300- and 360-rpms.	87

Advertiser/Product Index

Coming in June...

Japan's business TRON, FPU, and a special data-driven CPU

Plus

- **The ADSP-2100 DSP for graphics engine**
- **A logical design tool for relational databases**

FOR DISPLAY ADVERTISING INFORMATION, CONTACT:

Northern California and Pacific Northwest: Roy McDonald Assoc. Inc., 5915 Hollis St., Emeryville, CA 94608; (415) 653-2122.
Jim Olsen, P.O. Box 696, Hillsboro, OR 97123; (503) 640-2011.

Southern California and Mountain States: Richard C. Faust Co., 24050 Madison St., Suite 100, Torrance, CA 90505; (213) 373-9604.

Southwest: The House Co., 5252 Westchester, Suite 280, Houston, TX 77005; (713) 668-1007.

East Coast: Atlantic Representative Group, 349 Maple Place, Keyport, NJ 07735; (201) 739-1444.

New England: Arpin Associates, 40 Sterling St., Somerville, MA 02144; (617) 625-1777

Europe: Heinz J. Görgens, Parkstrasse 8a, D-4054 Nettetal 1 - Hinsbeck (F.R.G.); phone: (0 21 53) 8 99 88; telex 841(17)2153310=HJG tlx d.

For production information, conference, and classified advertising, contact Heidi Rex or Marian Tibayan.

IEEE MICRO, 10662 Los Vaqueros Cir., Los Alamitos, CA 90720; phone (714) 821-8380; fax (714) 821-4010.

	RS #	Page #
BOARDS		
Bus adapter	60	92
DSP/AIB board	44-45	90
Electronic image sensor	61	92
SCSI host adapter	35	89
CHIPS		
Data RAM	67	92
Interface chip	31-32	89
Microprocessor	40-42, 46-47	90
Processor	10, 43	87, 90
Protocol analyzer	30	89
Two-chip set	11-14	87
CONFERENCES & COURSES		
Hot Chips Symposium	2	4
DATA COMMUNICATIONS		
File manager	36-37	89
Interface controller	25-28	89
Internetwork router	38-39	89
Network camera control system	29	89
Network servers	66	92
Network system	64-65	92
Open-architecture software	34	89
MEMORY/STORAGE EQUIPMENT		
Backup system	80, 85	93
CD-ROM disk drive	33	89
Color disk	82	93
Data cartridge	81, 83	93
Magnetic disk	87	93
Tape controller	84	93
Tape drive	86	93
PUBLISHERS		
CS membership	—	2, 38
SOFTWARE		
CAD software package	22	88
Photodesign software	24	88
Preventive maintenance software	62	92
Simulation package	—	C.IV
Synthetic camera	20	88
SYSTEMS		
Binary compatibility architecture	15	87
Computer	54-56, 57-59	91
DSP link family	48-53	91
Project development tool	1	1
RISC family system	16-19	87
Security system	63	92
TEST & MEASUREMENT EQUIP.		
Finite element analysis	23	88
Vector engineering application	21	88

Continued from p. 96

Is the methane/hydrogen approach the only synthetic diamond-producing method under investigation?

Until now synthetic diamond has been produced at high temperatures and high pressures. These processes produce basically all of the synthetic diamond available today. Investigators of the low-pressure gas-synthesis technique are trying many different gases, ethane, propane, long-chain organics, and carbon monoxide. But the process use is basically the same: making diamond out of gaseous carbon. Other investigators use more exotic techniques to grow diamond. One group, J. Robertson et al. at the University of Houston, reports (in *Science,* Vol. 245, Feb. 1989) at least partial heteroepitaxy of diamond on silicon by an ion-beam deposition of carbon atoms. Heteroepitaxy is an extremely important step toward the possible use of diamond as a semiconductor.

Just how thin is diamond film?

From a few hundred to 10,000 atomic layers thick. If you were to look at a film, you probably would not see it at all, but if you were to touch it, you would feel it. For example, if you have a "no-stick" Teflon frying pan, you know that it feels a little slippery or greasy when you touch it.

This very thin film form of diamond must have several possible applications. Could you tell us what they might be?

Due to the film's low rate of light absorption, one possible use of the films is as windows between very sensitive X-ray detectors and the environment. These windows would let the X rays through but not the gases in the atmosphere, which could damage the detector. Another possible application is the coating of optical or infrared lenses for their protection from the water in the atmosphere. Because diamond absorbs only a very few frequencies of light, and because a very thin film will still be strong, these applications seem very promising.

Because of the film's hardness, it also would be useful in coating cutting tools and in protecting computer hard disks. If the disks and heads were covered with a diamond film, the magnetic head that reads the disk could be landed on the disk without endangering the data stored on the disk, instead of being shifted to a rest position on the side. And of course, due to the diamond's high heat conductivity, it would be very useful for electronics because the conductivity would easily keep the waste heat from building up.

How would diamond improve on the silicon used in semiconductor chips?

Diamond as a coating on silicon chips would mean we could use larger scale integration and smaller device spacings than those used presently. This is because the heat produced by the chip, which becomes a larger and larger concern with increased chip integration, is carried away faster by the diamond. If one can decrease the size of the devices used and increase the number of operations one chip can perform, the result would be an increase in the speed of operation. Since diamond as a semiconductor is faster than silicon, devices made in diamond would be faster than equivalent devices in silicon.

How fast could we expect diamond devices to operate?

If it is possible to grow diamond of a high enough quality to make it a semiconducting material, the device speed would ultimately be limited by the carrier mobility of the semiconductor. As the mobility of the holes in diamond is about three times that of holes in silicon, devices could be faster. However since growing diamond of a quality comparable to that of the silicon currently used will be difficult and expensive at best, diamond films, as a semiconductor, will probably always see very little application and only in very specialized environments.

What pressures are involved in the process?

Generally very low ones, about a hundredth to one tenth our normal pressure. We don't yet know what makes it possible for diamond to form at such low pressures. Most likely, what happens is that in this process we grow mostly the stable form, which is graphite, and also some diamond. Some reaction in the process then eats up the graphite, but the diamond is more stable and resists the attack. That's why the diamond remains and is able to grow. That's also the tricky part; if we're not careful, we get the graphite to grow instead.

What do we still need to know before diamond films can be used successfully in industry?

We need to understand how the film forms—that's crucial. We need to understand the process better and to control it better. We also need to understand how, if we have something more than just hydrogen and methane, we can incorporate other kinds of atoms.

Are you referring to doping?

Yes. If diamond is ever to be used as a semiconductor, we need to be able to controllably dope the diamond to make devices.

Doping diamonds seems to be proving practical in Japan. Will we see doping methods employed soon in the US?

Studying the doping of diamond as the next step for integrated diamond chip devices is a little premature. Using doping to study diamond growth is very appropriate and trying to dope diamond for some very specific applications may well be useful. It appears that the latter is what the Japanese are doing.

When can we expect products using diamond films to be manufactured here?

In limited, very special applications, diamond films are already on the market. I would guess in 1989 there will be X-ray windows and coatings on very specialized tools. Further out are the electronic applications. They are at least five to 10 years ahead. Even further out is the coating of plastics—near the year 2000. Furthest out of all is the use of large, single crystals of synthetic diamond as a semiconductor for integrated devices.

Reader Interest Survey

Indicate your interest in this department by circling the appropriate number on the Reader Interest Card.

Low 186 **Medium** 187 **High** 188

Department

Micro View

Diamond: A chip's best friend?

Lovely, sparkling, expensive diamonds may not be your idea of a useful industrial material. Yet, the imperfect, less-expensive types of diamonds have been used in this way for quite a long time.

Of current interest to semiconductor companies is research into the unusual qualities and rather startling new applications of artificially produced diamond films. This process appears so feasible that US researchers predict we will see electronic applications in five to 10 years, and other products even sooner. Right now, we can buy some coated products, particularly speaker tweeters from Sony that use low-grade diamond or diamond-like carbon (DLC) films. Another Japanese company, Sumitomo, is expected to release a chip that uses a diamond or DLC film as a protective layer for hostile environments like car engines.

Still, the diamond thin-film process remains something of a mystery. We need to understand how diamond film forms, how it bonds to a substrate, and how best to dope it with other substances before we can realize the most useful diamond applications.

To discover just where US research in the field is currently focused and how it is progressing, *IEEE Micro* contacted Stig B. Hagstrom. Hagstrom is chair of the Materials Science and Engineering Department at Stanford University and the director of its Center for Materials Research. He recently joined Stanford from Xerox's Palo Alto Research Center, where he served as manager of the Electronics Materials Laboratory.

Why is the diamond so attractive to chip manufacturers?

At room temperature diamond conducts heat better than any other material. It conducts heat five times better than copper, yet it is also one of the best electrical insulators, having a 5.5-eV band gap. Additionally, diamond is a faster semiconductor than silicon. For applications other than chips, such as computer hard disks that need to be protected, diamond is the hardest known material and has a friction coefficient similar to Teflon. These attractive qualities led scientists to think about ways to produce diamond films artificially and inexpensively.

Producing synthetic diamond for use in electronics is a recent development, isn't it? How did it come about?

Despite early research at Case Western Reserve in the US, it was Russian researchers in the 1960s who manufactured synthetic diamond in thin-film form using chemical vapor deposition techniques. They were pursuing a program to develop an inexpensive personal radiation monitor for use in case of a nuclear war. This approach proved uninteresting, but 10 years ago Japanese scientists demonstrated an improved process and US scientists became interested.

What form has your investigation taken?

My research group's investigation into diamond thin films is just getting started, although other groups at Stanford have been in this area for quite a while. The difference between our work and that at other universities is that we will be developing stronger ties to industry through increased communication and collaboration with interested companies. We intend the research to be primarily concerned with understanding the underlying science of the deposition rather than simply improving a given deposition process, although most researchers are pursuing this goal.

How do you create a diamond film?

To create a film, we place the material to be coated in a reactor with an inexpensive mixture of about one percent methane and 99 percent hydrogen. A chemical reaction is then caused by a variety of means—hot filament, microwave plasma, ion bombardment, among others—and the carbon in the methane forms diamond. If we do not carefully control the parameters of the reaction, we can easily form graphite or diamond-like carbon, which can be considered part graphite and part diamond. The substrate is most commonly silicon, but nickel, molybdenum, and gallium arsenide are also used. Eventually, we hope to be able to coat plastic, making a soft material with a very hard, scratch-resistant surface. This is not possible now because with current techniques the substrate must be heated to a temperature from 600 to 1,000 degrees Celsius, one much too hot for plastic.

Continued on p. 95

0272-1732/89/0400-0096$01.00 © 1989 IEEE

IEEE COMPUTER SOCIETY

A member society of the Institute of Electrical and Electronics Engineers, Inc.

Computer Society Offices

Headquarters Office
1730 Massachusetts Ave. NW
Washington, DC 20036-1903
Phone (202) 371-0101
Telex: 7180250437 IEEE COMPSO
Fax: (202)728-9614

Publications Office
10662 Los Vaqueros Cir.
Los Alamitos, CA 90720-2578
Membership and General Information: (714) 821-8380
Publication Orders: (800) 272-6657
Fax: (714) 821-4010

European Office
13, Ave. de L'Aquilon
B-1200 Brussels, Belgium
Phone: 32 (2) 770-21-98
Telex: 25387 AVVALB
Fax: 32 (2) 770-85-05

Asian Office
Ooshima Building
2-19-1 Minami-Aoyama, Minato-ku
Tokyo 107, Japan
Phone: 81 (3) 408-3118
Fax: 81 (3) 408-3553

Use the Reader Service Card to obtain information on:

- Membership application—student #203, others #202
- Perodicals subscription form for individuals #200
- Periodicals subscription form for organizations #199
- Publications catalog #201
- Standards working groups list #195
- Compmail+ international electronic mail/database brochure #194
- Technical committee list/application #197
- Chapters lists, start-up procedures—student/regular #193
- Student scholarship information #192
- Awards description/nomination forms #198
- Volunteer leaders/staff directory #196
- IEEE senior member application #204

Purpose

The IEEE Computer Society advances the theory and practice of computer science and engineering, promotes the exchange of technical information among 97,000 members worldwide, and provides a wide range of services to members and nonmembers.

Membership

Members receive the acclaimed monthly magazine *Computer*, discounts, and opportunities to serve (all activities are led by volunteer members). Membership is open to all IEEE members, affiliate society members, and others seriously interested in the computer field.

Publications and Activities

Computer. An authoritative, easy-to-read magazine containing tutorial and in-depth articles on topics across the computer field, plus news, conferences, calendar, interviews, and new products.

Periodicals. The society publishes six magazines and four research transactions. Refer to membership application or request information as noted above.

Conference Proceedings, Tutorial Texts, Standards Documents. The Computer Society Press publishes more than 100 titles every year.

Standards Working Groups. Over 100 of these groups produce IEEE standards used throughout the industrial world.

Technical Committees. More than 30 TCs publish newsletters, provide interaction with peers in specialty areas, and directly influence standards, conferences, and education.

Conferences/Education. The society holds about 100 conferences each year and sponsors many educational activities, including computing science accreditation.

Chapters. Regular and student chapters worldwide provide the opportunity to interact with colleagues, hear technical experts, and serve the local professional community.

European Office

Payments for Computer Society membership and publication orders are accepted by checks in Belgian, British, German, Japanese, Swiss, or US currency. Checks in US funds must be drawn on a US bank. Payment may also be made by American Express, Diners Club, Eurocard, Master Card, or Visa credit cards.

Asian Office

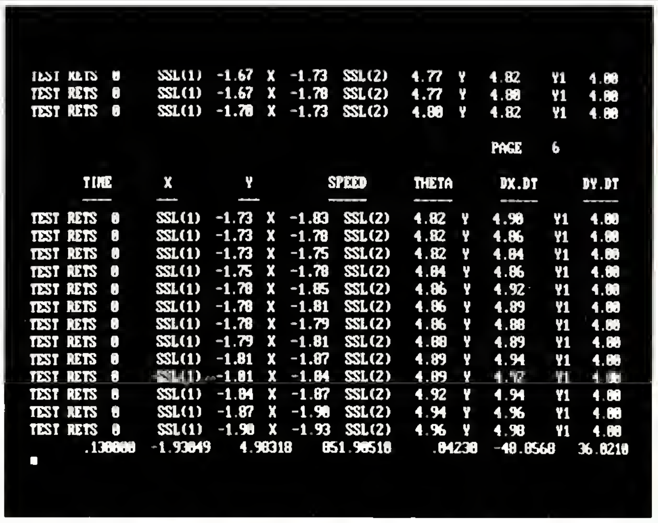
Payments for Computer Society membership and publication orders are accepted by checks in Japanese or US currency. Checks in US funds must be drawn on a US bank. Payment may also be made by electronic fund transfer to the Bank of Tokyo, Akasaka Branch, Toza acct. 0767956; the credit receiver is the IEEE Computer Society Headquarters Office. Payment may also be made by American Express, Diners Club, Eurocard, Master Card, or Visa credit cards.

Ombudsman

Members experiencing problems — magazine delivery, membership status, or unresolved complaints — may write to the ombudsman at the Publications Office.

SIMULATION BREAKTHROUGH

BEFORE



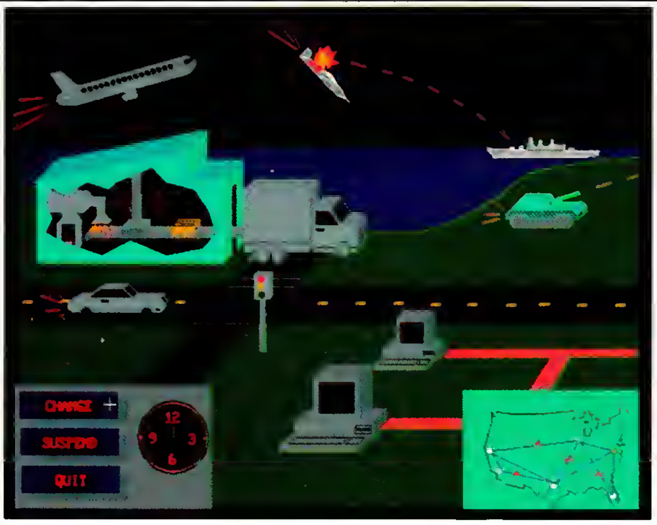
```
TEST RETS  0    SSL(1)  -1.67  X  -1.73  SSL(2)   4.77  Y   4.82    Y1   4.00
TEST RETS  0    SSL(1)  -1.67  X  -1.70  SSL(2)   4.77  Y   4.80    Y1   4.00
TEST RETS  0    SSL(1)  -1.70  X  -1.73  SSL(2)   4.80  Y   4.82    Y1   4.00

                                                            PAGE    6

        TIME        X          Y          SPEED      THETA      DX.DT      DY.DT
        ----       ---        ---         -----      -----      -----      -----
TEST RETS  0    SSL(1)  -1.73  X  -1.83  SSL(2)   4.82  Y   4.90    Y1   4.00
TEST RETS  0    SSL(1)  -1.73  X  -1.78  SSL(2)   4.82  Y   4.86    Y1   4.00
TEST RETS  0    SSL(1)  -1.73  X  -1.75  SSL(2)   4.82  Y   4.84    Y1   4.00
TEST RETS  0    SSL(1)  -1.75  X  -1.78  SSL(2)   4.84  Y   4.86    Y1   4.00
TEST RETS  0    SSL(1)  -1.78  X  -1.85  SSL(2)   4.86  Y   4.92    Y1   4.00
TEST RETS  0    SSL(1)  -1.78  X  -1.81  SSL(2)   4.86  Y   4.89    Y1   4.00
TEST RETS  0    SSL(1)  -1.78  X  -1.79  SSL(2)   4.86  Y   4.88    Y1   4.00
TEST RETS  0    SSL(1)  -1.79  X  -1.81  SSL(2)   4.88  Y   4.89    Y1   4.00
TEST RETS  0    SSL(1)  -1.81  X  -1.87  SSL(2)   4.89  Y   4.94    Y1   4.00
TEST RETS  0    SSL(1)  -1.81  X  -1.84  SSL(2)   4.89  Y   4.92    Y1   4.00
TEST RETS  0    SSL(1)  -1.84  X  -1.87  SSL(2)   4.92  Y   4.94    Y1   4.00
TEST RETS  0    SSL(1)  -1.87  X  -1.90  SSL(2)   4.94  Y   4.96    Y1   4.00
TEST RETS  0    SSL(1)  -1.90  X  -1.93  SSL(2)   4.96  Y   4.98    Y1   4.00
          .130000  -1.93049   4.98318   051.98510    .04230  -40.0560   36.0210
■
```

Difficult to understand results

AFTER



With SIMGRAPHICS you see your simulation in action. Draw your own icons or use one of ours.

Announcing new SIMSCRIPT II.5 with SIMGRAPHICS

Simulation models are now easier to build --results are easier to understand

Now you can provide the users of your SIMSCRIPT II.5 models with SIMGRAPHICS™ --graphical input and animation.

Results are easy to understand--an animated picture, histograms, pie charts and plots.

Because your results are understood, your recommendations are more likely to be acted upon.

Simulation simplified

SIMSCRIPT II.5 gives you a compact English-like language. Your simulation program reads like a description of the system you are studying. Animation is easy.

Your model development, checkout, modification and enhancement are greatly simplified.

Many successful applications

SIMSCRIPT II.5® is a well established, standardized, and widely used language with proven software support.

Typical applications include: military planning, manufacturing, communications, logistics, and transportation.

Free trial and training offer

The free trial contains everything you need to try SIMSCRIPT II.5 on your computer.

Try the SIMSCRIPT II.5 language, the quality and timeliness of our support, the accuracy of our documentation and the facilities for error-checking**--everything you need for a successful project.**

For 26 years CACI has offered trial use of its simulation software **--no cost, no obligation.**

Act now--free training offer

For a limited time we also include free training. Call today to avoid disappointment--class size is limited.

SIMSCRIPT II.5 is available on most popular computers including PC's running under OS/2 and the Mac II.

For immediate information

Call Doug Dittrich at (619) 457-9681, or FAX (619) 457-1184. In the UK, call Richard Eve on (01) 528-7980.

Rush information on SIMSCRIPT II.5 with SIMGRAPHICS free trial and training

Free trial-learn the reasons for the broad and growing popularity of SIMSCRIPT II.5.

No cost, no obligation.

Limited offer-return the coupon today and we will include one free course enrollment worth $950.

☐ Send information on your Special University Offer.

Name ______

Organization ______

Address ______

City ______ State ______ Zip ______

Telephone ______

Computer ______ Operating System ______

IEEE MICRO

Return to:
CACI Products Company
3344 North Torrey Pines Court
La Jolla, California 92037
Call Doug Dittrich at (619) 457-9681
FAX (619) 457-1184

In the UK
CACI Products Division
Regent House, 89 Kingsway
London WC2B 6RH, United Kingdom
Call Richard Eve on (01) 528-7980